TURING 图灵程序设计丛书

自制编程语言

[日] 前桥和弥 著

刘卓 徐谦 吴雅明 译



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

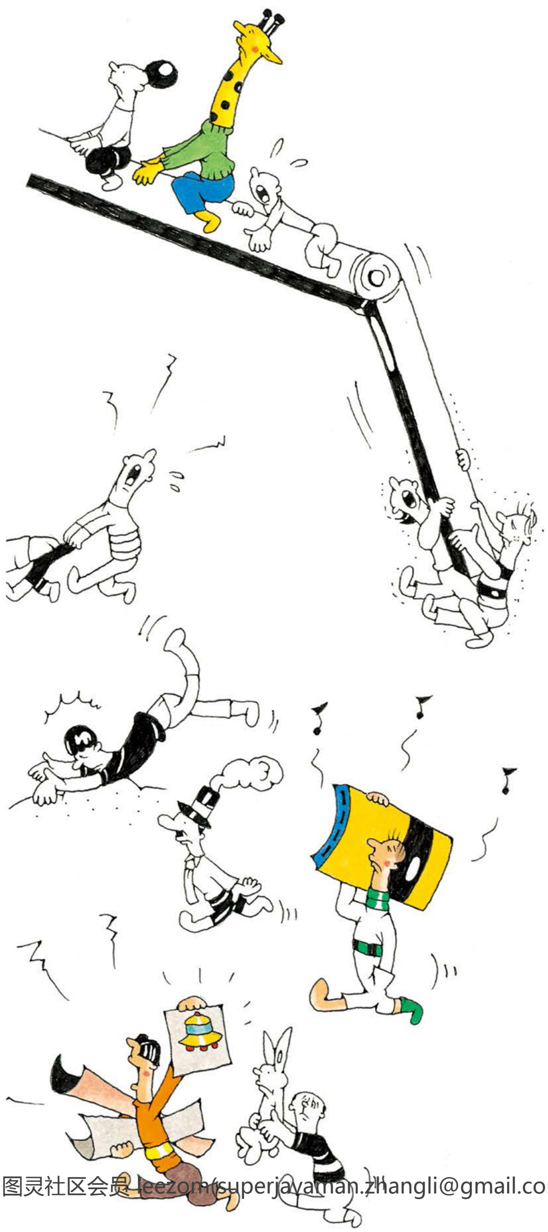
我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

前桥和弥（Maebasi Kazuya）

1969年出生，著有《征服C指针》、《彻底掌握C语言》、《Java之谜和陷阱》等。其一针见血的“毒舌”文风和对编程语言深刻的见地受到广大读者的欢迎。

作者主页：http://kmaebashi.com/。



TURING 图灵程序设计丛书

自制编程语言

[日] 前桥和弥 著
刘卓 徐谦 吴雅明 译



人民邮电出版社
北京

图书在版编目（CIP）数据

自制编程语言 / (日) 前桥和弥著；刘卓，徐谦，
吴雅明译. -- 北京：人民邮电出版社，2013.12（2016.10重印）
（图灵程序设计丛书）
ISBN 978-7-115-33320-9

Ⅰ. ①自… Ⅱ. ①前… ②刘… ③徐… ④吴… Ⅲ.
①C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字（2013）第238549号

内 容 提 要

本书手把手地教读者用C语言制作两种编程语言：crowbar与Diksam。crowbar是运行分析树的无类型语言，Diksam是运行字节码的静态类型语言。这两种语言都具备四则运算、变量、条件分支、循环、函数定义、垃圾回收等功能，最终版则可以支持面向对象、异常处理等高级机制。所有源代码都提供下载，读者可以一边对照书中的说明一边调试源代码。这个过程对理解程序的运行机制十分有帮助。

本书适合有一定基础的程序员和编程语言爱好者阅读。

◆ 著　　［日］前桥和弥
译　　刘　卓　徐　谦　吴雅明
责任编辑　乐　馨
执行编辑　金松月
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：24.75
字数：565千字　　2013年12月第1版
印数：8 001-8 300册　　2016年10月北京第4次印刷
著作权合同登记号　图字：01-2013-4381号

定价：79.00元
读者服务热线：(010)51095186转604　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第8052号

版权声明

（图灵公司感谢李典对本书的审读）

译者序

能翻开这本书的人，想必对编程都有着浓厚的兴趣。大部分编程爱好者都会利用业余时间写一些小程序、开源项目作为消遣，却很少有人会想要自己创造一门编程语言，这是为什么呢？

在翻译本书之前，如果别人问我要不要尝试自制编程语言，我一定会觉得他疯了。因为在潜意识里，我一直认为制作编程语言应该是 C 语言之父丹尼斯・里奇这样的业界大牛才能完成的浩大工程，作为一个普通程序员只要安于本分，用好已有的语言就已经足够了。

在翻译完本书后，我才发现自己真的是大错特错。原来创造一门编程语言，只需要一些 C 语言基础、一些正则表达式知识、加上不断思索的大脑就可以做到。如果你还觉得难以置信，那么就请看看在这本不算厚的书中，作者居然已经创造了两门编程语言，并且都具备高级编程语言的所有特性。

其实一开始的问题已经有了答案：很多看似难如登天的事情，一旦真的下决心去做，你会发现难度并没有想象中那么高，只是我们往往缺少一颗勇于挑战的心罢了。

本书记录了作者一步一步从零创造出编程语言的全过程，作者并不是什么行业精英，而是像你我一样的普通开发者。整本书中也没有用特别复杂的算法或酷炫的编程技巧，但是就凭借着一行行简单朴实的编程语句，作者最终完成了一个普通开发者看来几乎不可能完成的任务。阅读完本书后，除了自制编程语言的知识，我相信读者还能收获到一些更重要的东西。

本书原文讲到了日文编码的知识，为了更好的将内容精髓呈现给读者，我们大胆地将涉及日文编码的部分全部更改为中文编码的知识，译者刘卓还对此编写了很多原创的补充内容，力求能与原书保持同样的水平。如有错误或疏漏，还请读者随时指正。

读完全书后，你会对编程语言的原理和实现方式有一个全面深入的了解，比如你会明白为什么 Java 中 String 类型明明是对象类型却不能改变其内容，C 语言中为什么 `a++ + ++b` 这样看似合理的语句却会报错等。以前

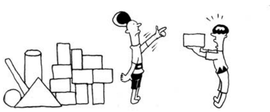
知其然而不知其所以然的问题都会得到答案，这对日后进行更高阶的开发有很大的帮助。

更重要的是，你可以获得自制编程语言的能力，从而可以去做很多以前敢想却没有能力做的事情，比如我现在就在构思能否创造一门以文言文和中国古代文化为基础的编程语言：易经八卦就是天然的二维矩阵，《九章算术》则有不少基础算法……相信读者还会有更加天才有趣的想法出现。如果能运用本书中的知识最终将其实现，那么这将是对翻译工作最好的肯定。

最后，在这里代表其他二位译者一并感谢在翻译过程中给予我们帮助和支持的家人、同事，让这本书最终得以问世。

徐谦

2013 年中秋



前言



这本书是为那些想独立制作一门编程语言的人而写的。

一听到这个话题，有的人会想：太疯狂了，制作编程语言肯定很有难度吧？有人会怀疑：制作编程语言能有什么用呢？其实这些都是误解。

制作编程语言在技术层面上其实并不难，只要掌握一些基础知识即可。而且，制作编程语言对于我们深入理解日常使用的 C、Java、JavaScript 等语言都有帮助。在一些应用程序的内置脚本语言中，我们也经常会因为种种限制从而萌生制作替代语言的想法。因此，自制编程语言并不是少数极客的个人癖好，它对大多数程序员都颇具实用价值。

日本关于制作编程语言的书已经很多了，其中一些还被选定为大学教科书。这些书中常出现有限状态机、NFA、LL(1)、LR(1)、SLA 等专业词汇，同时还大量使用∩、∊等数学符号，对于不熟悉这部分理论知识的人（包括我自己在内）来说非常难以读懂。针对这种现状，本书会偏重实践，避免枯燥的理论。

本书将分别制作两种编程语言：crowbar 与 Diksam。crowbar 是运行分析树的无类型语言，Diksam 是运行字节码的静态类型语言。无论哪种语言，都具备四则运算、变量、条件分支、循环、函数定义、垃圾回收等功能，最终版则可以支持面向对象、异常处理等高级机制。总之，作为现代编程语言所必须具备的功能都基本覆盖了（唯一可能没实现的就是多线程了吧）。所有源代码都提供下载，读者可以一边对照书中的说明一边调试源代码，这样应该不难理解整个程序的运行机制。

当然，要一次实现如此多功能的编程语言，对于初学者而言可能有点吃力，因此本书会详细介绍 crowbar、Diksam 的制作步骤，请放心。

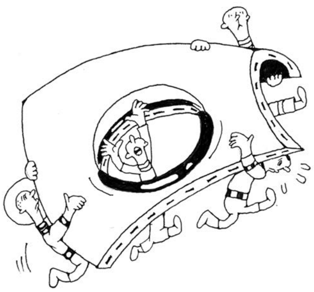
在制作编程语言的过程中，我体会到了一种无法用语言形容的快乐。其实无论在日本或其他地区，世界上还有很多人都在尝试自制编程语言，这正是编程语言不断增加的原因。如果以本书为契机，有朝一日你也向本已混乱的巴比伦之塔再添一门新语言的话，作为本书作者，这将是无上的光荣。

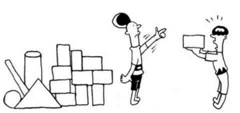


在本书的撰写过程中，得到了很多朋友的帮助与支持：

感谢百忙之中通读原稿并给出很多改进意见的吉田敦、间野健二、藤井壮一、山本将；感谢对本书原型，即网页版“自制编程语言”提出意见的朋友；感谢对博客连载“自制编程语言日记”提出意见的读者朋友，以及实际使用 crowbar 与 Diksam 并提出意见的朋友。最后还要感谢每次对我延迟交稿仍然充满耐心的技术评论社的熊谷裕美子编辑。多亏大家的鼎力支持，本书才终能完成，在此我表示深深的谢意。

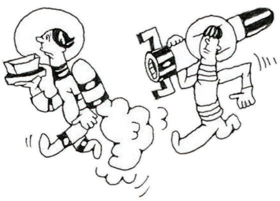
2009 年 5 月 7 日 01:06 J.S.T
前桥和弥

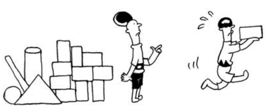




目录 CONTENTS

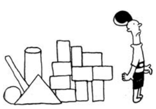










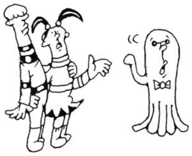


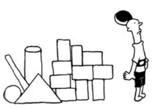


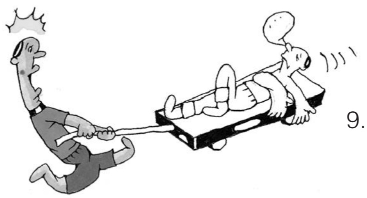






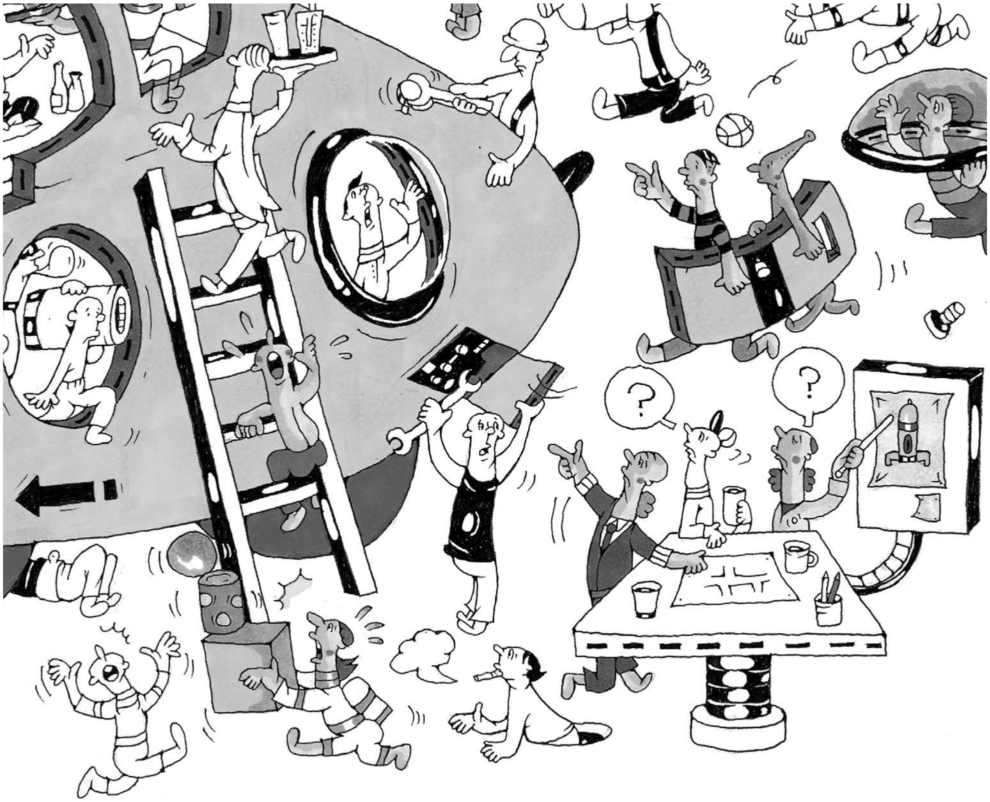












第 1 章

引子

1.1 为什么要制作编程语言

本书的主题是自制编程语言。单说现在被广泛使用的编程语言，就有 C、C++、Java、C#、Perl、Python、Ruby、PHP、Lisp、JavaScript 等。可能有人会质疑，既然已经有这么多语言了，真的有必要再特意创造一门新的语言吗?

实际上，自制编程语言还是大有益处的。

1. 可以帮助理解编程语言的内部运行机制

编程语言是程序员每天都要使用的工具。深刻地理解这个工具，对程序员来说非常重要。

一般来说，重新编写一个与已有程序相似的程序会被说成是“重复发明轮子”，这在行业内是不被认同的。但本书中想要实现的，偏偏是在众多语言存在的前提下再制作一门新的语言，正是“重复发明轮子”。这是深刻理解编程语言的最佳途径（缺点是要花很多时间）。

2. 能制作领域专用语言

比如在 Unix 的世界中，有 sed 和 awk 两种历史悠久的专为文本处理定制的语言（后来在此方向上发展出了 Perl 语言）。PHP 则是专门面向 Web 程序开发的语言。如果掌握了制作编程语言的技术，就可以在必要的情况下制作出领域专用语言（DSL，Domain-Specific Language）。

领域专用语言不一定会像 Perl 与 PHP 那么复杂，在很多情况下，如果能书写条件分支或者简单语句的话会方便许多，这也可以看作是一种专用领域。

比如在业务流程处理等软件中，很多时候为了切换测试环境与生产环境的数据库，需要重写配置文件，而这一操作经常会引发问题（比如由于版本升级需要增加配置文件项目，此时必须与旧版本配置文件合并）。这时候我们可能就会想，如果能直接在配置文件中写 `if` 语句将其按域名分开就好了。

除此以外，我们在填写数据时可能希望能支持类似 Excel 的简单算术公式，在玩游戏时希望能把游戏中的对话导出到一个外部文件中，等等。这些都可以看作专用领域并制作对应的 DSL。

3. 可以用编程语言扩展应用程序

将以上两方面的考量进一步延伸，我们就会得到以通用语言扩展某个应用

程序的构想。Emacs 这个编辑器就内置了 Emacs Lisp 这种 Lisp 方言，从而为 Emacs 的自定义提供了无限的可能性。同理，Microsoft Office 也可以使用 VBA 进行扩展。

对于这类应用程序扩展语言，当然完全可以使用某种已有的编程语言（Lua 等就在向这个方向发展），也可以在编写应用程序时从底层到扩展全部自己实现。这样就无需担心使用其他编程语言在版本升级时引起的兼容性问题了。

4. 说不定还会变成名人

如果自制的编程语言能在世界范围内得到广泛使用，那就太棒了。比如 Ruby 之父松本行弘先生就是世界名人。

不过坦白讲，通过自制编程语言来获得成功实在是太难了。即便语言被创造出来，如果没人用的话就不会产生相应的软件，这样就更不会有人用了。况且，即便真的因为发明了新的语言而变成了名人，通过这个赚到钱的希望也十分渺茫啊。其实我自己最近写的语言处理器都是免费发布的（不这样的话，语言没法普及呀）。

5. 自制编程语言非常有趣

啰嗦了这么多，说到底其实是因为自制编程语言非常有趣。

自制一门编程语言确实是一件非常有意思的事。有人说过“想写出终极程序的程序员，最终都去写操作系统或者编程语言了”，你可以通过自制编程语言感受到接触最核心技术的乐趣。

让尽可能多的人感受到这种乐趣，这正是本书的目标。



1.2 自制编程语言并不是很难

一提起自制编程语言，很多人都会觉得这是一件非常难的事情。

比如，即便是一个很常见的赋值语句：

```
a1 = b1 + b2 * 0.5;
```

在自制编程语言时都必须考虑到以下几个要点。



1. 需要将 a1、b1、b2 作为变量名解析出来。如果按照 C 语言的语法规则，变量名只能由字母或下划线开头，从变量名第二个字符开始才允许出现字母或数字。所以首先必须扫描这个语句，然后将匹配上述语法规则的部分提取出来。
2. 0.5 是一个含有小数点的常量，在提取这类常量时，能否用“数字组合 + 小数点 + 数字组合”来概括所有常量的特征呢（还要考虑是否允许 00.10 这样的数值）。当然我们的提取规则还要能处理 2 这样不含小数点的数值。
3. 乘法运算符 * 比 + 拥有更高的运算优先级，语句必须被解析为 b1 + (b2 * 0.5)。
4. b2 * 0.5 的计算结果，必须在与 b1 进行加法运算前就应该取得。也就是说对于复杂的计算，需要保存很多类似这样的临时运算结果。

假如你已经有了一定的编程经验，肯定能想到上面这些难点，甚至可以说你的编程经验越丰富，就越能感受到这其中隐藏着极大的难题。

不过，编程语言的语言处理器在 FORTRAN 诞生后已经经过了多年的研究，上面的这些难点都已经可以从前人那里找到解决方法 *。

* 当然，在早年原始的研发条件下，人们为了开发第一个编程语言编译器还是花费了相当大的精力，据说实现初版的 FORTRAN 编译器所花费的工时，累计达到了 216 人月 [1]。

在本书中，上面 1 ～ 3 的问题会用到名为 yacc 及 lex 的工具。问题 1 和问题 2 用 lex，问题 3 通过 yacc 解决。yacc 和 lex 都是非常老的工具了，现在流行的 LL 语言大多内置了 yacc。可能有人会说：“既然是以学习为目的去制作一门编程语言，如果还使用工具的话就太投机取巧了吧。”（这话很有道理。）所以在本书中，也会稍微介绍一下不使用这些工具的解决方法。

无论是使用工具，还是基于一些已有的解决方案自己编写，如果能掌握一些窍门的话，自制编程语言其实并不难。

那么你想不想试试自己制作一门编程语言呢？自己创造编程语言这件事情，不管怎么说都是很酷的吧。

1.3　本书的构成与面向读者

本书由以下的章节构成：

- 第 1 章　引子
- 第 2 章　试做一个计算器
- 第 3 ~ 4 章　制作无类型语言 crowbar
- 第 5 章　中文支持与 Unicode

- 第 6 ~ 8 章　制作静态类型的语言 Diksam
- 第 9 章　应用篇

第 1 章即是你正在阅读的章节。本章会对全书的构成以及讲解方式进行说明。

第 2 章通过制作一个简单的计算器，介绍 yacc/lex 的基本使用方法。其实讲解 yacc/lex 的部分，选择“计算器”为例实在有点老套，但确实没有比这更合适的题目了。此外还会介绍如何不依赖 yacc，使用递归下降分析器（Recursive Descent Parser）来制作一个计算器。

从第 3 章开始，会实际制作有一定行数规模的编程语言。

3 ～ 4 章会制作一个名为 crowbar 的无类型解释型语言，6 ～ 8 章则主要制作名为 Diksam 的支持静态类型的编译型语言（名字的由来会在后文提到）。在第 5 章中，会针对使用编程语言时的中文支持与 Unicode 问题进行说明。

第 9 章阐释闭包（Closure）及异常处理机制等进阶功能。

本书将使用 C 语言作为编程语言的语言处理器（编译器、解释器等）的编写语言（理由见后文中的具体说明）。而 crowbar 与 Diksam 最终都会累积为具备一定行数规模的程序（crowbar 约 8000 行，Diksam 约 2 万行）。

因此，阅读本书的读者最好具备两个条件：

1. 已经会 C 语言
2. 具备阅读较长代码的能力

不过无论哪个条件都不是必须的。

对于**条件 1**需要说一点的是，Java、C++、C# 等都是从 C 语言发展出来的语言，所以对于已经学习过这些语言的人来说，读 C 语言代码不会特别吃力。像预处理程序、指针等 C 语言特有的知识，建议你借此机会一并学习一下。因为至少就现阶段来说，无论是专家还是业余爱好者，但凡是程序员都免不了要用到 C 语言。而在 crowbar 或 Diksam 中，并没有使用很多 C 语言特有的功能。比如说不会出现 `*p++` 这种不易理解的写法，更多是写成数组下标的形式。

对于**条件 2**要说的是，虽然一个语言处理器整体来看是个上规模的程序，但是其基础构成的部分并不会很庞大。本书不会对每一行代码逐一进行注释，而是侧重于介绍解决问题的思路，所以如果仅仅是想阅读一下本书的话，是不需要具备阅读较长代码的经验的。但若你最后不满足于书中的讲解，还想要自己去阅读一下 crowbar 或者 Diksam 源代码的话，因为代码行数很多，编程经验尚浅的朋友

读起来可能会有压力。不过无论是业界还是外界人士，作为程序员总有一天会接触到大规模代码的程序，将本次实践作为入门的第一步也不是一件坏事。

综上所述：

如果你觉得自己不是本书所面向的读者，想办法加入其中不就行了？

所以无需担心什么，门槛其实没有你想的那么高。凡是对语言处理器有兴趣的朋友都是本书面向的读者。

1.4　用什么语言来制作

如前文所述，本书将使用 C 语言作为语言处理器的编写语言。

都什么年代了还用 C 语言？可能会有人这样想吧。其实就连我自己也会这样想。

但本书还是使用了 C 语言，其中一个理由是因为 yacc/lex 都是面向 C 语言的工具。

yacc/lex 本身是很老的工具。老工具虽然都有一些历史遗留问题，但也有其优点，即正是因为历史悠久，所以会积累下更详尽的技术文档。如前文所述，目前的 LL 语言大多使用 yacc。

另一个使用 C 语言的理由是：想要降低“依赖程度”的话，C 语言是最适合的。

比如说用 Java 编写软件，运行环境中必须安装 JVM（Java 虚拟机）。如果用 C# 则必须要安装 .NET Framework。在自制编程语言的理由中，我们曾经列举了“可以用编程语言扩展应用程序”这一条，并且提到，如果能在编写应用程序的时候从底层到扩展全部自己实现会更加放心，其目的就是为了不依赖 JVM 或 .NET Framework。这样在 Java 或 .NET 版本升级时也就无需操心了。

此外考虑到组合各种应用程序这个用途，C 语言在众多编程语言中可以说是最具通用性的。无论被组合的应用程序采用何种语言编写，毫无疑问都可以调用 C 语言。

1.5 要制作怎样的语言

1.5.1 要设计怎样的语法

编程语言有很多种，C、C++、Java、C# 等都是**面向过程的编程语言**（C++、Java、C# 虽然也被称为面向对象，但可以把面向对象看作是面向过程的一个派生）。目前看来，虽然面向过程的语言是主流，但还存在 Haskell、ML 这样的**函数式编程语言**。函数式编程语言就是“变量值无法被更改”的一种语言 *。

* 从这个定义来说，Lisp 严格讲还不能算是函数式编程语言。

对于已经习惯了面向过程语言的人来说，肯定会想“变量值无法更改还怎么写程序呀”。其实这类语言已经编写出了很多实用的程序。在函数式编程的基础上发展出了如 Prolog 这样的**逻辑编程语言**以及被称为**并行程序设计语言**的 Erlang。

不过目前被广泛使用的仍然是面向过程的编程语言，本书中的代码示例使用的也都是面向过程的语言风格，当然里面还会加入面向对象的一些功能实现。在本书中，除了会有 C++、Java、C# 这种基于类的面向对象之外，也会涵盖类似 JavaScript 这种没有类的面向对象。

语法层面上，会使用类似 C 语言的风格。crowbar 的示例代码如代码清单 1-1 所示，Diksam 的示例代码如代码清单 1-2 所示。

代码清单 1-1
crowbar 版 FizzBuzz

```
for (i = 1; i < = 100; i++) {
    if (i % 15 == 0) {
        print("FizzBuzz\n");
    } elsif (i % 3 == 0) {
        print("Fizz\n");
    } elsif (i % 5 == 0) {
        print("Buzz\n");
    } else {
        print("" + i + "\n");
    }
}
```

代码清单 1-2
Diksam 版 FizzBuzz

```
int i;

for (i = 1; i < = 100; i++) {
```

```
        if (i % 15 == 0) {
            println("FizzBuzz");
        } elsif (i % 3 == 0) {
            println("Fizz");
        } elsif (i % 5 == 0) {
            println("Buzz");
        } else {
            println("" + i);
        }
    }
```

顺便说一下这个名为 FizzBuzz 的小程序，其运行机制如下：

> 输出从1到100的数字，如果为3的倍数时，则将数字替换为Fizz，5的倍数时则输出Buzz，同时为3与5的倍数时输出FizzBuzz。

这个小程序引自下面的文章。文章大意是建议企业在面试程序员时，至少应聘者能写出这种程度的代码再考虑录用。

> ◎为什么自称程序员的人写不出程序?
> http://www.aoky.net/articles/jeff_atwood/why_cant_programmers_program.htm

看了示例就能明白，无论 crowbar 还是 Diksam，都是与 C 语言非常类似的语言。

如上所述，本书虽然会创造一门新语言但仍然会用到 C 语言，所以本书所面向的读者应该是已经掌握了 C 语言的（还没有掌握的人可以先去学习一下）。因此如果选择 C 语言风格的语法，读者应该会感到很亲切，更重要的是笔者本人已经习惯了 Java、C# 这种以 C 语言为基础的编程语言。

C 语言是很老的语言了，这门语言不是在前期经过严谨的设计，而是在项目中一边实践一边慢慢发展起来的，因此语法上难免有很多考虑不周的地方。比如在 C 语言中赋值使用 `=`，即数学中的等号。而 C 程序员在初学者阶段编写 `if` 语句时，肯定免不了会写成这样：

```
if (a = 0) {    ←应该写"=="但是写成了"="
      ⋮
}
```

这样惨痛的教训至少也要经历一次吧。赋值在 Pascal 等语言中，一般使用 `:=`。如果让一个没有编程经验的人来学习，Pascal 这种语法应该更加友好一些。

不过我现在是要制作一门新的编程语言，而使用这门新语言的人应该都已经习惯了 C 语言的运算符，如果这里将赋值运算符定为 `:=` 的话反而会引起混乱，

说不定我自己就先头晕了。所以经验之谈是，语法上的些许优劣还是要给“习惯”让步的。

——出于这种考虑，我最终决定制作一门与 C 语言类似的编程语言。

决定语法风格是编程语言创造者的特权。如果顾虑用户习惯，可以参考并整合已有的编程语言。当然，也可以完全不考虑用户的感受，去创造一门“理想的语言”。虽然我是以 C 语言的语法为基础，但还是想到了以下几点可以改进的地方。

1. `if`条件在C语言中，如果按条件执行的语句只有一句，则{}可以省略。但是这经常会造成混乱，很多项目的编码规范中都会规定必须包含{}。因此最好在语法层面直接将{}设置为不可省略（crowbar、Diksam均如此）。
2. 既然已经将`if`条件中的{}设置为不可省略，那么`if`后面的()要怎么办呢？（关于这一点，我起初在crowbar中尝试了一下省略`if`的括号，结果发现在crowbar中()是不可省略的。）
3. 伴随着语言的逐步完善，考虑到要增加一些关键字（参考2.3.1节的补充知识），此时再处理与已存在程序的变量名相冲突的问题就比较麻烦，所以考虑在所有的变量前加上`$`（Perl或PHP等的解决方式），或者将关键字全部以大写字母开头（Modula-2等的解决方式）。
4. `switch case`语句中，最好能去掉忘了写`break`就会进入下一个`case`这种容易产生问题的设计（Java没有改进这一点，C#则做了一些半吊子的改进）。
5. `switch case`语句中，如果没有进入任何一个`case`条件分支，也没有写`default`分支，那么在运行时直接报错会不会更好一些（Pascal就是这样处理的）？
6. 编码规范通过缩进来约束怎么样？比如像Python那样通过缩进来表明逻辑结构。
7. 对于我来说，阅读Python风格的代码还有些吃力，因此是不是做成像C语言那样用花括号包裹语法块、把强制缩进的检查交给编译器去做比较好呢？

我希望读者朋友们也能够用好语言开发者的特权，不断去追求“更加理想的语言”。呃，虽然我这样讲可能会被说成是站着说话不腰疼吧。

1.5.2 要设计怎样的运行方式

程序员中应该无人不知，编程语言有编译型语言和解释型语言两种。

编译型语言中，C 和 C++ 比较有代表性。这类语言通常会将程序员编写的程序源代码，最终输出为机器码的可执行文件。

但是想要输出机器码的话，必须首先掌握机器码才行。即便学习了机器码并

写出了编译器，该编译器也无法输出供其他型号 CPU 运行的文件 *。

* 为了解决这个问题，一般的编译器都会将依赖 CPU 生成的机器码的部分单独归为一个名为后端的模块，根据不同的 CPU 可以更换相应的后端，就可以支持其他型号的 CPU 了。

这类生成机器码的编程语言的优点是运行速度非常快，但是编译器性能优化的相关技术，学习起来非常有难度。另外，在自制编程语言的理由中曾经列举了“可以用编程语言扩展应用程序”这一点，而输出机器码的编译器并不适合这个用途。因此本书中会选择解释型语言。

虽说“解释型语言”只是一个词，但是其实现方法又分很多种。

解释型语言的“解释”一词源自英语的interpreter，是“能进行翻译的物体”的意思。编译器将源代码翻译为机器码，之后 CPU 直接运行机器码就可以了。与此相对的解释型语言，则将程序员编写的源代码通过解释器这一程序一边解析一边运行——这种公式化的定义看起来只有简单的两个步骤，但现实中几乎不存在这么单纯的解释型语言（DOS 的批处理脚本或 UNIX 的 SHELL 脚本是最接近解释型语言的定义的）。虽说名为“解释型语言”，但其中的大多数都会将源代码临时转换为某种中间形态。

比如有代码清单 1-3 这样的代码。

代码清单 1-3
简单的 if 语句 *

```
if (a == 10) {
    printf("hoge\n");
} else {
    printf("piyo\n");
}
```

* 代码中的 hoge、piyo 这两个单词，经常在输出无意义的语句时使用（多见于日本，英语国家则较多使用 foo、bar）。详细请参考以下的页面：http://avnpc.com/pages/devlang#hoge

从机器的角度看，源代码其实只是一些文字的排列组合而已，机器是无法直接运行的。现在大多数编程语言，都会将代码转换成一种叫分析树（parse tree，也叫语法分析树或语法树）的东西。上面的代码如果做成分析树，则如图 1-1 所示。

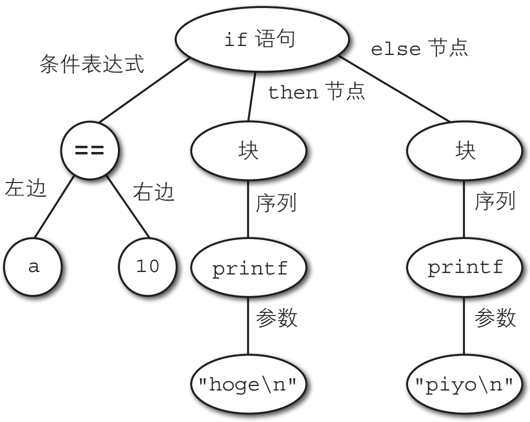


图 1-1
分析树示例

Perl、Ruby 等语言，一旦将代码转换为分析树后，分析树将无法再还原回源代码。

本书第 2 章以后所用到的语言 crowbar 就是采用这种运行方式的语言。

对于这类语言来说，从源代码到分析树的构建过程还是得称为“编译”。但是这里的编译器是在程序启动时自动执行的。由于分析树会生成在内存里，因此不会生成目标代码或目标文件，所以程序员（用户）一般意识不到有编译器在执行。这类语言如果存在语法错误，会在刚开始运行时就被报出来，这正是源代码被一次性全部读入并构建分析树的证明。如果是纯粹的解释型语言，如批处理脚本或 SHELL 脚本，则会运行到有语法错误的地方才会报错。

那么，相对于 Perl、Ruby 这样的运行分析树型语言，在 Java 等语言中，取代分析树的则是更底层的字节码，然后通过解释器运行字节码。字节码只是一些简单的数字排列，为了尽可能地让人读懂字节码，字节码中的所有指令都被加上了一些名为助记符（mnemonic）的字符，代码清单 1-3 的源代码经过这样一番处理之后最终会变成代码清单 1-4 的样子（源代码中的 `printf` 改为 `System.out.println`，并使用 javap 输出）。

代码清单 1-4
Java 的字节码

```
 0: bipush 10
 2: istore_1
 3: iload_1
 4: bipush  10
 6: if_icmpne       20
 9: getstatic
12: ldc
14: invokevirtual
17: goto    28
20: getstatic
23: ldc
25: invokevirtual
```

本书第 5 章以后所用到的语言 Diksam，就是采用这种运行方式的语言。

在 Java 中，编译器生成的字节码会被保存在 class 文件中。但是在 Diksam 中，编译器会在程序启动时执行，因此字节码保存于内存中，不会生成类似 class 文件的东西。由此可以看出，从用户的角度出发，不需要意识到 Diksam 内部其实有字节码在执行。Python 也是使用了类似的处理机制。

补充知识　“用户”指的是谁？

前文曾写道“因此程序员（用户）一般意识不到有编译器在执行。”

通常来说，用户是指使用用程序员编写的程序的人，但是在这里，因为我们是要制作一门编程语言，所以本书中的用户应该是指使用我们制作的编程语言的人，即程序员。

这种指代在操作系统、类库、编程语言等面向程序员的文档中经常出现，不过可能有读者会有误解，在此特别补充说明一下。

补充知识　解释器并不会进行翻译

在很多入门书中，提到编译器与解释器时，一般会采用以下说明：

> **编译器会将源代码一次性全部翻译为机器码。**
> **与此相对的解释器，不会事先做一次性翻译，而是在运行的同时，逐行分块地将源代码翻译为机器码。**

请允许我说句老实话，这样的说明是完全错误的。

解释器会将源码或分析树解析为字节码这种中间形态，并且一边解析一边运行，但是解释器并不会将源码翻译为机器码。

Java 或 .NET Framework 都具备在运行的同时将字节码转换为机器码的功能，这叫作“JIT（Just-In-Time）编译”技术，而这部分技术并不属于解释器。

那么解释器具体是如何运行程序的呢？读到后面你就会明白了。

1.6 环境搭建

1.6.1 搭建开发环境

本书的开发语言是 C 语言，辅助工具是 yacc 和 lex。

UNIX（包含 Linux 等）大部分都已经预装了开发所需的 yacc 和 lex，当然也有例外，而 Windows 则默认没有预装。不过无需担心这些，我们完全可以全部使用自由软件来搭建一个可用的开发环境。

那么，下面我们就开始介绍这些软件的获取途径。

1. C 编译器

免费的 C 编译器可以使用 GNU 项目提供的 GCC（GNU Compiler Collection）。

Linux 等免费的 UNIX 环境下大多都预装了 GCC*。Windows 下可以使用 MinGW（Minimalist GNU for Windows）。

* 最近 Linux 不预装 GCC 的情况似乎越来越多了。

可以从下面的 URL 下载。

```
http://www.mingw.org/download.shtml
```

安装 MinGW 时，UNIX 环境下的程序会将构建（build）时使用到的 make 工具也一并安装。不过，安装完毕后可执行文件名有点奇怪，是 `mingw32-make.exe`，我将其复制并重命名为 `gmake.exe` 以方便使用。

2. cygwin 或 MSYS

cygwin 是可以运行在 Windows 上的类 UNIX 环境。比如说想在命令行提示符中列出当前文件夹内的文件时，Windows（DOS）会使用 `DIR` 指令，UNIX 则使用 `ls` 指令。一般用惯了 UNIX 的人，往往会在 Windows 的命令行提示符中不自觉地敲出 `ls` 却尴尬地发现指令不存在，而安装了 cygwin 就可以避免这样的情况发生。那么对于不经常使用 UNIX 的人还有必要装 cygwin 吗？因为在后文中提到的 bison 要使用 UNIX 中的 m4 工具，所以无论是 cygwin 还是 MSYS，至少还是要安装其中一个的 *。MSYS 与 cygwin 都是在 Windows 上模拟 UNIX 环境的软件。

* m4 其实也可以单独安装，但似乎没有独立的安装包，可能会非常麻烦。

cygwin 可以从下面的网址中获取：

```
http://cygwin.com/
```

MSYS 可从 MinGW 页面中下载：

```
http://www.mingw.org/download.shtml
```

此外，因为 cygwin 也包含 GCC，可以没有 MinGW 而通过 cygwin 安装 GCC。但是使用 cygwin 安装的 GCC 编译，运行时需要依赖 cygwin1.dll 文件，在其他机器运行还需要把 DLL 也复制过去，所以还是使用 MinGW 更方便。

3. bison

如果环境无法直接运行 yacc，可以使用 GNU 项目提供的 bison。

```
http://gnuwin32.sourceforge.net/packages/bison.htm
```

4. flex

同理，如果环境无法直接运行 lex，可以使用 lex 的免费版 flex。

```
http://gnuwin32.sourceforge.net/packages/flex.htm
```

补充知识　关于 bison 与 flex 的安装

bison 由 GNU 项目提供。GNU 项目是由理查德·斯托曼（Richard Matthew Stallman）创立的项目，目标在于建立一个完全相容于 UNIX 的自由软件环境。

GNU 项目提供的软件的许可证为 GPL（通用公共许可协议，General Public License）。粗略地说，GPL 是这样一种许可证：

- **发行 GPL 的程序时，必须公开源代码并且声明源代码的出处；**
- **包含 GPL 源代码的程序，必须受 GPL 许可证条款约束；**
- **程序即使以动态链接方式使用 GPL 程序，也必须受 GPL 许可证条款约束。不过这个限制在 LGPL 许可证（Lesser GPL）中有所放宽。**

也就是说，你的程序中只要用到 GPL 的程序，哪怕这部分再小，你的程序也会自动变成 GPL 程序，必须与源代码同时公开。这对于那些为了防止盗版而不得不采取一些措施的商用软件来说简直是致命的。因此也有人戏称 GPL 的这个特性是“GPL 传染”或“GPL 病毒”。

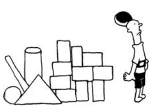
那么 bison 是否也是如此呢？后文会有说明，bison 的作用是将用户编写的配置文件输出为 C 语言格式的代码。这里的 C 代码中会包含一些属于 bison 的代码。那么是不是说使用 bison 去制作编程语言，所做出的编程语言在发行上也必须遵守 GPL 许可证呢？关于 bison 输出的 C 代码这一点，是 GPL 的一个特例，可以不受 GPL 许可证约束。此处在 GNU 项目有关 GPL 的 FAQ 页面中有如下的记载：

> **碰巧的是，Bison 也可以用于开发非自由软件。这是因为我们明确允许在 Bison 的输出结果中包含的 Bison 的标准解析程序可以不受限制。我们做此决定，是因为已经存在与 Bison 类似的工具被用于非自由软件的开发。**
> **http://www.gnu.org/licenses/gpl-faq.ja.html**

另一方面，flex 则是遵循 BSD 许可证（Berkeley Software Distribution，加州大学伯克利分校开发的软件套件集合）的（不是修订版 BSD）。BSD 许可证的程序再次发行时，文档中必须要附加 BSD 的版权信息。

flex 会像 bison 一样输出 C 代码，这里的 C 代码也像 bison 一样，会包含一些属于 flex 的代码。但是这部分代码并不需要附加 BSD 的版权信息。因为 flex-2.5.34 携带的 COPYING 文件中有这样的描述：

Note that the “flex.skl” scanner skeleton carries no copyright notice. You are free to do whatever you please with scanners generated using flex; for them, you are not even bound by the above copyright.

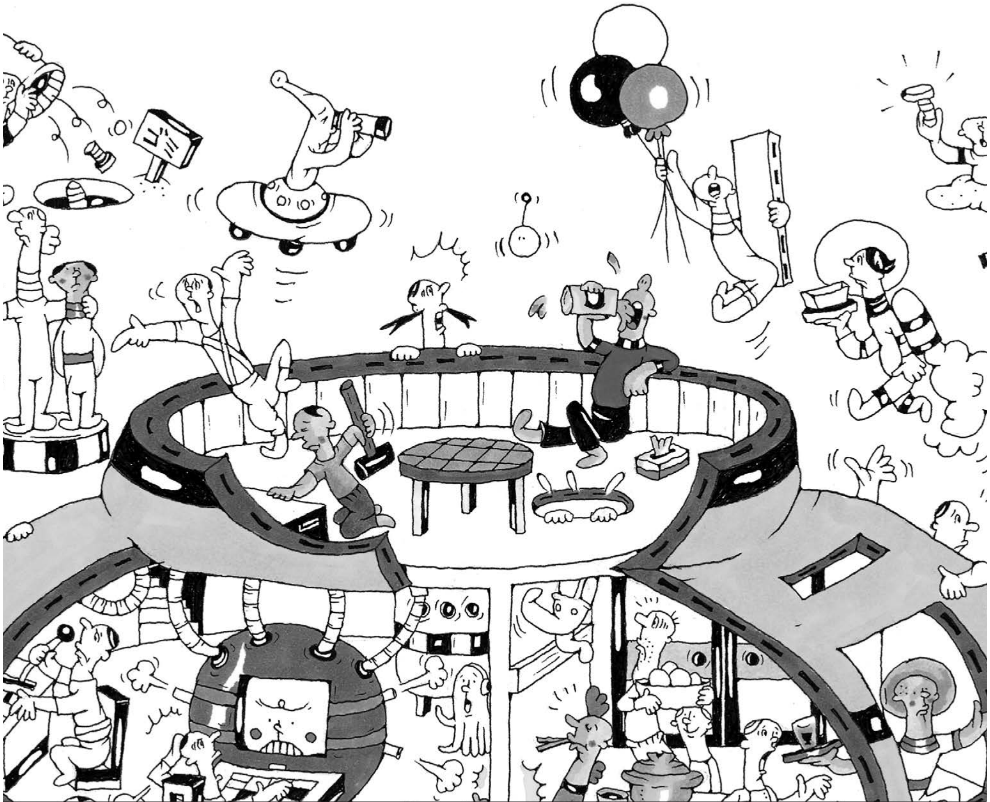


1.6.2 本书涉及的源代码以及编译器

本书所涉及的源代码，可以在作者的网站上下载：

```
http://avnpc.com/pages/devlang#download
```

在开始撰写本书之前，crowbar 和 Diksam 就已经存在一些公开的版本了，本书所用到的代码都对其进行了重新的整理和修正，因此本书相关的代码将重新以 `book_ver` 作为版本号。比如本书最开始制作的 crowbar 的版本号就是 crowbar book_ver.0.1。



第 2 章

试做一个计算器

2.1　yacc/lex 是什么

如前文所述，本书会使用 yacc 和 lex 这两个工具。本章中将利用 yacc/lex 尝试编写一个简单的计算器程序。

一般编程语言的语法处理，都会有以下的过程。

1. 词法分析

将源代码分割为若干个**记号**（token）的处理。

2. 语法分析

即从记号构建**分析树**（parse tree）的处理。分析树也叫作**语法树**（syntax tree）或**抽象语法树**（abstract syntax tree，AST）*。

* 严格讲，包含代码中所有记号的叫作分析树或语法树，将一些无用记号剔除的才叫作抽象语法树，本书中并没有特意区分。

3. 语义分析

经过语法分析生成的分析树，并不包含数据类型等语义信息。因此在语义分析阶段，会检查程序中是否含有语法正确但是存在逻辑问题的错误。

一般来说执行语义分析时主要会做数据类型的解析以及错误检查，但本书中使用的 crowbar 语言并没有设置变量类型，因此也不会进行数据类型的检查，所以 crowbar 并不存在一个明确的语义分析阶段（Diksam 中是存在这个阶段的，位于 fix_tree.c 源文件中，请参考本书 6.3.4 节）。

4. 生成代码

如果是 C 语言等生成机器码的编译器或 Java 这样生成字节码的编译器，在分析树构建完毕后会进入代码生成阶段。

比如说有如下的代码：

```
if (a == 10) {
    printf("hoge\n");
} else {
    printf("piyo\n");
}
```

执行词法分析后，将被分割为如图 2-1 所示的记号（每一个块就是一个记号）：

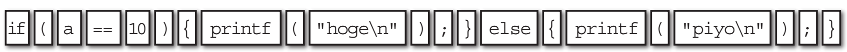
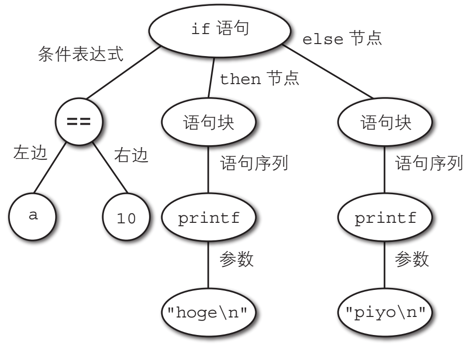


图 2-1
分割为记号

对此进行语法分析后构建的分析树，如图 2-2 所示（同图 1-1）。

图 2-2
if 语句的分析树



执行词法分析的程序称为**词法分析器**（lexical analyzer）*。lex 的工作就是根据词法规则自动生成词法分析器。

* 也称为扫描器（lexer 或 scanner）。

执行语法分析的程序则称为**解析器**（parser）。yacc 就是能根据语法规则自动生成解析器的程序。

顺便说一下，yacc 是“Yet Another Compiler Compiler”的缩写。顾名思义，Compiler Compiler 就是生成编译器的编译器。yacc 其实只能生成编译器的一部分（解析器部分），却自称编译器的编译器，未免有些名不副实。而在 yacc 诞生时还因为存在其他几个编译器的编译器，所以 yacc 的作者干脆取“又一个（Yet Another）编译器的编译器”之意，起名为 yacc*。lex 则只是简单地取自 lexical analyzer。

* 最近这样的例子还有 Ruby 的 VM 的 YARV，即“Yet Another Ruby VM”的缩写，还有 YAML 起初也是叫作“Yet Another Markup Language”。

yacc 和 lex 一起使用，可以将一个特殊格式的定义文件输出为 C 语言代码。

因为两者都是很老的工具，所以数据传递都采用全局变量的方式，现在看起来实在有些简陋。即便如此，至少 yacc 仍然作为 Perl、Ruby 等语言的语言处理器活跃在第一线。词法分析器则相对简单，完全可以自己编写，所以正式的编程语言一般都不会使用 lex。

yacc 与 lex 在 UNIX 的标准环境下大多都已经预装了，在 Windows 等环境一般没有预装，所以需要使用其免费的替代品 bison 和 flex（获得方法请参考 1.6.1 节）。

补充知识　词法分析器与解析器是各自独立的

如前文所述，源文件的语法分析，首先要经过词法分析器分割为记号，然后才经过解析器做语法分析，是这样一个分工合作的过程。

常常有人会对于 C 或 Java 提出这样的疑问：

```
a+++++b;
```

这行代码明明可以理解为 `a++ + ++b`，为什么编译器还会报错呢？

正是因为有了前文所说的词法分析器与解析器的分工合作机制，所以产生错误也就不难理解了。因为词法分析器先于语法分析器运行，在词法分析阶段还无法获得 C 或 Java 的语法规则，代码就会被分割成 `a ++ ++ + b`，从而导致报错。

2.2 试做一个计算器

向不了解 yacc/lex 的人介绍其功能的话，与其一上来就举“用 yacc/lex 制作编程语言”的例子，不如先从一些简单的例子讲起比较好。所以我们以一个简单的“计算器”为例做介绍。

将计算器作为 yacc/lex 的示例实在有些老套，但是又很实用。Windows 都会预装一个带有图形界面的计算器软件，仔细想来，PC 上明明有好用的键盘，却还要用鼠标去一个一个点计算器上的按钮未免有些傻[①]。正因为 Windows 的计算器不好用，所以有很多人会选择使用普通的计算器。可明明眼前就摆着高性能的电脑，偏要用买来的计算器，同样也显得很奇怪，更不要说还有“附带计算器功能的鼠标垫”这种创意产品，就更加本末倒置了。无论是 Windows 自带的计算器，还是从日杂店买来的计算器，都从上面看不到运算符号的优先顺序，也无法直接计算带括号的式子（因为看不到前一个输入的值）。几十个数值求和时，你会不会担心万一中间输错了该怎么办？我经常会。

而我们要制作的计算器，会通过命令行方式启动，可以通过键盘输入整个算式，然后直接显示计算结果。因此可以直观地看到运算的优先顺序，比如输入

```
1 + 2 * 3
```

会得到结果 7 而不是 9。因为能看到整个算式，所以还可以很容易地检查有没有输错。

计算器的指令名为mycalc。一个实际运行的例子是这样的（`%` 是命令提示符）：

① Windows 的计算器支持使用键盘输入，但是有很多初级用户并不知道这个功能，仍然用鼠标点击。此外一些迷你键盘去掉了数字键盘区域，在上面使用计算器很不方便。——译者注

```
% mycalc        ←启动 mycalc
1+2             ←输入算式
>>3.000000      ←显示结果
1+2*3           ←乘法与加法的混合运算
>>7.000000      ←按运算优先顺序输出结果
```

虽然只用了整数，却输出“3.000000”这样的结果，这是因为 `mycalc` 在内部完全使用 `double` 进行运算。

2.2.1 lex

lex 是自动生成词法分析器的工具，通过输入扩展名为 `.l` 的文件，输出词法分析器的 C 语言代码。而 flex 则是增强版的 lex，而且是免费的。

词法分析器是将输入的字符串分割为记号的程序，因此必须首先定义 `mycalc` 所用到的记号。

`mycalc` 所用到的记号包括下列项目：

- 运算符。在 mycalc 中可以使用四则运算，即 `+`、`-`、`*`、`/`。
- 整数。如 `123` 等。
- 实数。如 `123.456` 等。
- 换行符。一个算式输入后，接着输入换行符就会执行计算，因此这里的换行符也应当设置为记号。

在 lex 中，使用**正则表达式**定义记号。

为 `mycalc` 所编写的输入文件 mycalc.l 如代码清单 2-1 所示。

代码清单 2-1
mycalc.l

```
%{
#include <stdio.h>
#include "y.tab.h"

int
yywrap(void)
{
    return 1;
}
%}
%%
"+"             return ADD;
"-"             return SUB;
"*"             return MUL;
```

```
15: "/"             return DIV;
16: "\n"            return CR;
17: ([1-9][0-9]*)|0|([0-9]+\.[0-9]+) {
18:     double temp;
19:     sscanf(yytext, "%lf", &temp);
20:     yylval.double_value = temp;
21:     return DOUBLE_LITERAL;
22: }
23: [ \t] ;
24: . {
25:     fprintf(stderr, "lexical error.\n");
26:     exit(1);
27: }
28: %%
```

代码第 11 行为 `%%`，此行之前的部分叫作**定义区块**。在定义区块内，可以定义初始状态或者为正则表达式命名（即使不知道正则表达式的具体内容也可以命名）等。在本例中放置了一些 C 代码。

第 2 行至第 9 行，使用 `%{` 和 `%}` 包裹的部分，是想让生成的词法分析器将这部分代码原样输出。后续程序所需的头文件 `#include` 等都包含在这里，比如第三行用 `#include` 包含进来的 y.tab.h 头文件，就是之后 yacc 自动生成的头文件。下面的 `ADD`、`SUB`、`MUL`、`DIV`、`CR`、`DOUBLE_LITERAL` 等都是在 y.tab.h 中用 `#define` 定义的宏，其原始出处则定义于 mycalc.y 文件中（详见 2.2.3 节）。这些宏将记号的种类区分开来。顺便附上表 2-1 说明记号的具体含义。

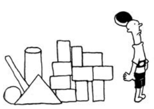
表 2-1
记号及其含义

记号	含义
ADD	加法（addition）运算符 +
SUB	减法（subtraction）运算符 -
MUL	乘法（multiplication）运算符 *
DIV	除法（division）运算符 /
CR	回车符（carriage return）
DOUBLE_LITERAL	double 类型的字面常量（literal）

第 5 行到第 9 行定义了一个名为 `yywrap()` 的函数。如果没有这个函数的话，就必须手动链接 lex 的库文件，在不同环境下编译时比较麻烦，因此最好写上。本书不会再对这个函数做深入说明，简单知道其作用，直接使用就可以了。

第 12 行到 27 行是**规则区块**。读了代码就能明白，这一部分是使用正则表达式 * 去描述记号。

* 关于 lex 的正则表达式，请参考 2.2.2 节。



在规则区块中遵循这样的书写方式：一个正则表达式的后面紧跟若干个空格，后接 C 代码。如果输入的字符串匹配正则表达式，则执行后面的 C 代码。这里的 C 代码部分称为动作（action）。

在第 12 ～ 16 行中，会找到四则运算符以及换行符，然后通过 `return` 返回其特征符。所谓特征符，就是上文所述在 y.tab.h 中用 `#define` 定义、用来区别记号种类的代号。

之前提到了这么多次“记号”，其实我们所说的记号是一个总称，包含三部分含义，分别是：

1. **记号的种类**
   比如说计算器中的 `123.456` 这个记号，这个记号的种类是一个实数（`DOUBLE_LITERAL`）。
2. **记号的原始字符**
   一个记号会包含输入的原始字符，比如 `123.456` 这个记号的原始字符就是 `123.456`。
3. **记号的值**
   `123.456` 这个记号代表的是实数 123.456 的值的意思。

对于 + 或 - 这样的记号来说，只需要关注其记号种类就可以了，而如果是 `DOUBLE_LITERAL` 记号，记号的种类与记号的值都必须传递给解析器。

第 17 行的正则表达式，是一个匹配“数值”用的正则表达式。表达式匹配成功的结果，即上面列举的记号三要素中，“记号的原始字符”会在相应动作中被名为 `yytext` 的全局变量（这算是 lex 的一个丑的设计）引用，并进一步使用第 19 行的 `sscanf()` 进行解析。

动作解析出的值会存放在名为 `yylval` 的全局变量中（又丑一次）。这个全局变量 `yylval` 本质上是一个联合体（union），可以存放各种类型记号的值（在这个计算器程序中只有 `double` 类型）。联合体的定义部分写在 yacc 的定义文件 mycalc.y 中。

到第 28 行，又出现了一次 `%%`。这表示规则区块的结束，这之后的代码则被称为**用户代码区块**。在用户代码区块中可以编写任意的 C 代码（例子中没有写）。与定义区块不同，用户代码区块无需使用 `%{ %}` 包裹。

2.2.2 简单正则表达式讲座

lex 通过正则表达式定义记号。正则表达式在 Perl、Ruby 语言中广泛使用，而 UNIX 用户也经常会在编辑器或 grep 命令中接触到。本书的读者可能未必都有这样的技术背景，所以我们在本书涉及的范围内，对正则表达式做一些简单的说明。

在代码清单 2-1 的第 17 行中有这样一个正则表达式（初看可能稍微有点复杂）：

```
([1-9][0-9]*)|0|([0-9]+\.[0-9]+)
```

首先，[与] 表示匹配此范围内的任意字符。而 [ ] 还支持使用连接符的缩写方式。比如写 1-9 与写 123456789 是完全一样的。

最初圆括号中的 [1-9] 代表匹配 1 ～ 9 中任意一个数字。其后的 [0-9] 代表匹配 0 ～ 9 的任意一个数字。

在此之后的 *，代表匹配前面的字符 * 0 次或多次。

* 因为后面还会有 [0-9]* 这样的写法，所以严格讲，“匹配前面的字符”其实是“匹配前面的表达式”。

因此，[1-9][0-9]* 这个正则表达式，整体代表以 1 ～ 9 开头（只有 1 位），后接 0 个以上的 0 ～ 9 的字符。而这正是我们在 mycalc 中对整数所做的定义。

在表 2-2 中列举了若干字符串，并展示其是否匹配我们所制定的整数规则。

表 2-2
字符串的匹配

字符串	是否为整数	备注
5	是	5 匹配 [1–9]
310	是	3 匹配 [1–9]，后面的 10 匹配 [0–9]*
012	否	开头的 0 不匹配 [1–9]
0	否	开头的 0 不匹配 [1–9]

如上表所示，mycalc 不会将 012 这样的输入作为数值接收，这完全符合我们的预期。

但是将 0 也排除的话还是有问题的，程序必须能接受所有的实数。因此在正则表达式中又使用了 |。| 代表“或”的意思。

第 17 行的正则表达式就被修正为：

```
([1-9][0-9]*)|0|([0-9]+\.[0-9]+)
```

现在应该可以明白前半段正则表达式的整体意思了，即将整数（0 以外）的正则表达式与 0 通过 | 分成两部分然后并列（加上圆括号 () 是将这部分作为一个集合才能通过 | 与 0 并列）。

后半部分的 `[0-9]+\.[0-9]+` 中用到了 `+`。`*` 是代表“匹配前面的字符 0 次或多次”，而 `+` 则是“匹配前面的字符 1 次或多次”，因此这部分整体代表“0 ～ 9 的数字至少出现一次，后接小数点 `.` 后又接至少一位 0 ～ 9 数字”。这些与前面整合起来，共同构成了 `mycalc` 对于实数的定义。

小数点的书写不是只写一个 `.` 而是写成了 `\.`，因为 `.` 在正则表达式中有特殊的含义（后文即将介绍），所以需要使用 `\` 转义。`[ ]`、`*` 、`+`、`.` 等这些在正则表达式中有特殊含义的字符称为**元字符**，元字符可以像上文那样用 `\` 或双引号[①]进行转义。代码的第 12 ～ 14 行，就是使用双引号转义的方法对乘法和加法的运算符进行了定义。

第 23 行的正则表达式 `[ \t]` 是对空格以及制表符进行匹配，对应动作为空，因此可以忽略每一行的空白字符。

第 24 行的 `.` 会匹配任意一个字符。这里用于检测是否输入了程序不允许的字符。

首先，lex 将输入的字符串分割到若干个记号中时，会尽可能选择较长的匹配。比如 C 语言中同时有 + 运算符和 ++ 运算符，那么当输入 ++ 时，lex 不会匹配为两个 + 运算符，而是返回一个 ++（如果不按这个逻辑，程序很难正常工作）。如果两个规则出现同样长度的匹配时，会优先匹配前一个规则。也就是说，如果输入字符直到最后一条规则（匹配任意字符）才匹配成功的话，说明这个字符不符合前面所有的规则，是错误的输入。

在表 2-3 中列举了一些常用的元字符。元字符以外的字符直接书写就可以了。

表 2-3
lex 中正则表达式的元字符

`*`	匹配 0 个或者多个前面的字符
`+`	匹配 1 个或者多个前面的字符
`.`	匹配任意 1 个字符
`[abc]`	匹配 a 或 b 或 c
`[a-c]`	匹配 a ～ c 的字符
`[^a-c]`	匹配 a ～ c 以外的字符
`""`	被包裹的字符不会被作为元字符，而是匹配其字面含义
`\`	转义后面的元字符

① 正则表达式有很多不同的风格，lex 所使用的风格支持使用双引号转义元字符，而常用的 PCRE 风格则只支持反斜线转义。——译者注

2.2.3 yacc

yacc 是自动生成语法分析器的工具，输入扩展名为 `.y` 的文件，就会输出语法分析器的 C 语言代码。bison 则是 GNU 项目所发布的 yacc 的功能扩充版。

`mycalc` 中 yacc 的输入文件 mycalc.y 如代码清单 2-2 所示。

代码清单 2-2
mycalc.y

```
%{
#include <stdio.h>
#include <stdlib.h>
#define YYDEBUG 1
%}
%union {
    int          int_value;
    double       double_value;
}
%token <double_value>      DOUBLE_LITERAL
%token ADD SUB MUL DIV CR
%type <double_value> expression term primary_expression
%%
line_list
    : line
    | line_list line
    ;
line
    : expression CR
    {
        printf(">>%lf\n", $1);
    }
expression
    : term
    | expression ADD term
    {
        $$ = $1 + $3;
    }
    | expression SUB term
    {
        $$ = $1 - $3;
    }
    ;
term
    : primary_expression
    | term MUL primary_expression
    {
        $$ = $1 * $3;
```

```
39:     }
40:     | term DIV primary_expression
41:     {
42:         $$ = $1 / $3;
43:     }
44:     ;
45: primary_expression
46:     : DOUBLE_LITERAL
47:     ;
48: %%
49: int
50: yyerror(char const *str)
51: {
52:     extern char *yytext;
53:     fprintf(stderr, "parser error near %s\n", yytext);
54:     return 0;
55: }
56:
57: int main(void)
58: {
59:     extern int yyparse(void);
60:     extern FILE *yyin;
61:
62:     yyin = stdin;
63:     if (yyparse()) {
64:         fprintf(stderr, "Error ! Error ! Error !\n");
65:         exit(1);
66:     }
67: }
```

第 1 ～ 5 行与 lex 相同，使用 `%{%}` 包裹了一些 C 代码。

第 4 行有一句 `#define YYDEBUG 1`，这样将全局变量 `yydebug` 设置为一个非零值后会开启 Debug 模式，可以看到程序运行中语法分析的状态。我们现在还不必关心这个。

第 6 ～ 9 行声明了记号以及**非终结符**的种类。正如前文所写，记号不仅需要包含种类，还需要包含值。记号的值可能会有很多类型，这些类型都声明在联合体中。本例中为了方便说明，定义了一个 `int` 类型的 `int_value` 和 `double` 类型的 `double_value`，不过目前还没有用到 `int_value`。

非终结符是由多个记号共同构成的，即代码中的 `line_list`、`line`、`expression`、`term` 这些部分。为了分割非终结符，非终结符最后都会以一个特殊记号结尾。这种记号称作**终结符**。

第 10 ～ 11 行是记号的声明。mycalc 所用到的记号种类都在这里定义。ADD、SUB、MUL、DIV、CR 等记号只需要包含记号的种类就可以了，而种类为 DOUBLE_LITERAL 的记号，其种类被指定为 <double_value>。这里的 double_value 是来自上面代码中 %union 联合体的一个成员名。

第 12 行声明了非终结符的种类，并指明了这些非终结符的值在联合体中对应的成员名。

与 lex 一样，13 行的 %% 为分界，之后是**规则区块**。yacc 的规则区块，由**语法规则**以及 C 语言编写的相应动作两部分构成。

在 yacc 中，会使用类似 BNF（巴科斯范式，Backus Normal Form）的规范来编写语法规则。

计算器程序因为规则部分中混杂了动作，阅读起来有点难度，所以在代码清单 2-3 中，仅仅将规则部分抽出，并加入了注释。

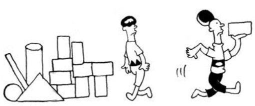
代码清单 2-3
计算器的语法规则

```
line_list  /* 多行的规则 */
    : line /* 单行 */
    | line_list line /* 或者是一个多行后接单行 */
    ;
line  /* 单行的规则 */
    : expression CR  /* 一个表达式后接换行符 */
    ;
expression /* 表达式的规则 */
    : term  /* 和项 */
    | expression ADD term /* 或 表达式 + 和项 */
    | expression SUB term /* 或 表达式 - 和项 */
    ;
term  /* 和项的规则 */
    : primary_expression  /* 一元表达式 */
    | term MUL primary_expression  /* 或 和项 * 一元表达式 */
    | term DIV primary_expression  /* 或 和项 / 一元表达式 */
    ;
primary_expression  /* 一元表达式的规则 */
    : DOUBLE_LITERAL  /* 实数的字面常量 */
    ;
```

为了看得更清楚，可以将语法规则简化为下面的格式：

```
A
 : B C
 | D
 ;
```

即 A 的定义是 B 与 C 的组合，或者为 D。



第 1 ～ 4 行的书写方式，是为了表示该语法规则在程序中可能会出现一次以上。在 mycalc 中，输入一行语句然后敲回车键后就会执行运算，之后还可以继续输入语句，所以需要设计成支持出现一次以上的模式。

另外，请注意在上面的计算器的语法规则中，语法规则本身就包含了运算符的优先顺序以及结合规律。如果不考虑运算符的优先顺序（乘法应该比加法优先执行），上文的语法规则应该写成这样。

```
expression  /* 表达式的规则 */
    : primary_expression  /* 一元表达式 */
    | expression ADD expression  /* 或 表达式 + 表达式 */
    | expression SUB expression  /* 或 表达式 - 表达式 */
    | expression MUL expression  /* 或 表达式 * 表达式 */
    | expression DIV expression  /* 或 表达式 / 表达式 */
    ;
primary_expression  /* 一元表达式的规则 */
    : DOUBLE_LITERAL  /* 实数的字面常量 */
    ;
```

那么在这样的语法规则下，yacc 是如何运作的呢？我们以代码清单 2-3 为例一起来看看吧。

大体上可以这样说，yacc 所做的工作，可以想象成一个类似“俄罗斯方块”的过程。

首先，yacc 生成的解析器会保存在程序内部的栈，在这个栈中，记号就会像俄罗斯方块中的方块一样，一个个堆积起来。

比如输入 1 + 2 * 3，词法分析器分割出来的记号（最初是 1）会由右边进入栈并堆积到左边。



像这样一个记号进入并堆积的过程，叫作**移进**（shift）。

mycalc 所有的计算都是采用 double 类型，所以记号 1 即是 DOUBLE_LITERAL。当记号进入的同时，会触发我们定义的规则：

```
primary_expression
    : DOUBLE_LITERAL
```

然后记号会被换成 primary_expression。



类似这样触发某个规则并进行置换的过程，叫作**归约**(reduce)。

primary_expression 将进一步触发规则：

```
term
    : primary_expression
```

然后归约为 term。



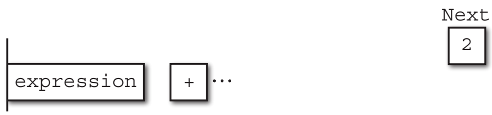
再进一步根据规则：

```
expression
    : term
```

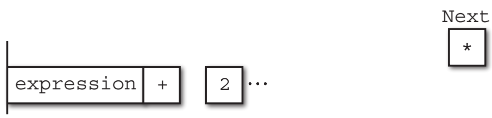
最终被归约为一个 expression。



接下来，记号 + 进入。在进入过程中，由于没有匹配到任何一个规则，所以只好老老实实地进行移进而不做任何归约。



接下来是记号 2 进入。



经过上述同样的规则，记号 2（DOUBLE_LITERAL）会经过 primary_expression 被归约为 term。



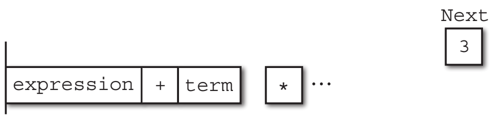
这里记号 2 本应该匹配到如下的规则：

```
expression
    | expression ADD term
```

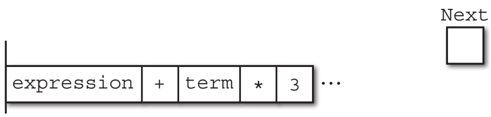
yacc 和俄罗斯方块一样，可以预先读取下一个要进入的记号，这里我们就可以知道下一个进入的会是`*`，因此应当考虑到记号 2 会匹配到`term`规则的可能性。

```
term
    | term MUL primary_expression
```

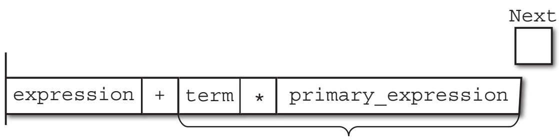
归约完毕后再一次移进。



接下来记号 3 进入，



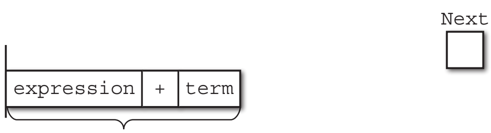
被归约为`primary_expression`后，



`term`、`*`、`primary_expression`这一部分将匹配规则：

```
term
    | term MUL primary_expression
```

被归约为`term`。



之后，expression、+、term 又会匹配规则

```
expression
    | expression ADD term
```

最终被归约为 expression。



每次触发归约时，yacc 都会运行该规则的相应动作。比如乘法对应执行的规则如下文所示。

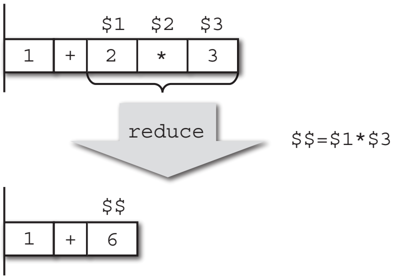
```
term
        | term MUL primary_expression
        {
            $$ = $1 * $3;
        }
```

动作是使用 C 语言书写的，但与普通的 C 语言又略有不同，掺杂了一些 $$、$1、$3 之类的表达式。

这些表达式中，$1、$3 的意思是分别保存了 term 与 primary_expression 的值。即 yacc 输出解析器的代码时，栈中相应位置的元素将会转换为一个能表述元素特征的数组引用。由于这里的 $2 是乘法运算符（*），并不存在记号值，因此这里引用 $2 的话就会报错。

$1 与 $3 进行乘法运算，然后将其结果赋给 $$，这个结果值将保留在栈中。在这个例子中，执行的计算为 2 * 3，所以其结果值 6 会保留在栈中（如图 2-3）。

图 2-3
归约发生时栈的动作



咦，$1 与 $3 对应的应该是 term 和 primary_expression，而不是 2 与 3 这样的 DOUBLE_LITERAL 数值才对呀，为什么会作为 2 * 3 来计算呢？

可能会有人提出上面的疑问吧。这是因为如果没有书写动作，yacc 会自动

补全一个 { $$ = $1; } 的动作。当 DOUBLE_LITERAL 被归约为 primary_expression、primary_expression 被归约为 term 的时候，DOUBLE_LITERAL 包含的数值也会被继承。

```
term
        : primary_expression
        {
            $$ = $1;    /* 自动补全的动作 */
        }
 ⋮
primary_expression
        : DOUBLE_LITERAL
        {
            $$ = $1;    /* 自动补全的动作 */
        }
```

$$ 与 $1 的数据类型，分别与其对应的记号或者非终结符的类型一致。比如，DOUBLE_LITERAL 对应的记号被定义为：

```
%token <double_value>      DOUBLE_LITERAL
```

expression、term、primary_expression 的类型则为：

```
%type <double_value> expression term primary_expression
```

这里的类型被指定为 <double_value>，其实是使用了在 %union 部分声明的联合体中的 double_value 成员。

由于我们以计算器为例，计算器的动作会继续计算得出的值，但仅靠这些还不足以制作编程语言。因为编程语言中都会包含简单的循环，而语法分析只会运行一次，所以动作还要支持循环处理同一处代码才行。

因此在实际的编程语言中，会从动作中构建分析树。这部分处理的方法会在后面的章节中介绍。

2.2.4 生成执行文件

接下来，让我们实际编译并链接计算器的源代码，生成执行文件吧。

在标准的 UNIX 中，按顺序执行下面的指令（% 是命令行提示符），就会输出名为 mycalc 的执行文件。

```
% yacc -dv mycalc.y    ←运行 yacc
```

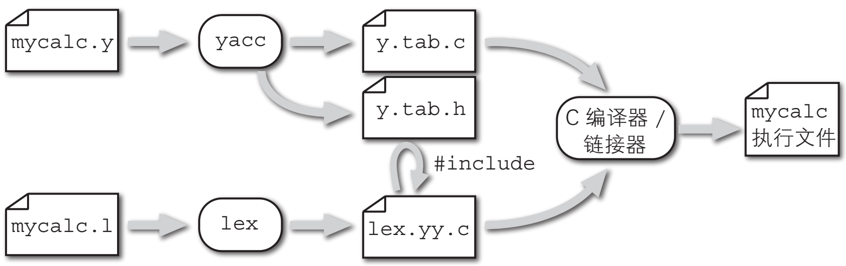
```
% lex mycalc.l            ←运行 lex
% cc -o mycalc y.tab.c lex.yy.c    ←使用 C 编译器编译
```

如果在 Windows 的环境下，参考 1.6.1 节中的说明，需要安装 gcc、bison、flex，然后运行下面的指令（`C:\Test>` 是命令行提示符）。

```
C:\Test>bison --yacc -dv mycalc.y        ←用 bison 代替 yacc 并运行
C:\Test>flex mycalc.l                    ←运行 flex
C:\Test>gcc -o mycalc y.tab.c lex.yy.c   ←使用 C 编译器编译
```

这个过程中会生成若干文件。其流程以图片表示的话，如图 2-4 所示。

图 2-4
yacc/lex 的编译



y.tab.c 中包含 yacc 生成的语法分析器的代码，lex.yy.c 是词法分析器的代码。为了将 mycalc.y 中定义的记号及联合体传递给 lex.yy.c，yacc 会生成 y.tab.h 这个头文件。

此外，作为 C 语言程序当然要有 `main()` 函数，在 mycalc 中 `main()` 位于 mycalc.y 的用户代码区块（第 49 行以后），最终编译器会负责合并代码，所以这里的 `main()` 与其他 .c 文件分离也不要紧。在 `main()` 函数中的全局变量 `yyin` 可以设定输入文件，调用 `yyparse()` 函数。

由于我们使用 bison 替代了 yacc，默认生成的文件就不是 y.tab.c 和 y.tab.h，而是 mycalc.tab.c 和 mycalc.tab.h。所以在上例中添加了 `--yacc` 参数，可以让 bison 生成与 yacc 同名的文件。本书为了统一，bison 会始终带上 `--yacc` 参数。

2.2.5 理解冲突所代表的含义

实际用 yacc 试做一下解析器，可能会被**冲突**（conflict）问题困扰。所谓冲突，就是遇到语法中模糊不清的地方时，yacc 报出的错误。

比如 C 语言的 `if` 语句，就很明显有语法模糊问题。

```
if (a == 0)
```

```
    if (b == 0)
        printf(在这里 a 与 b 都为 0\n);
else
    printf(这里是 a 非 0？错！\n);
```

上面的代码中，我们不清楚最后的 else 对应的究竟是哪一个 if，这就是冲突。

yacc 运行时，遇到下面任意一种情况都会发生冲突。

- 同时可以进行多个归约。
- 满足移进的规则，同时又满足归约的规则。

前者称为归约 / 归约（reduce/reduce）冲突，后者称为移进 / 归约（shift/reduce）冲突。

即便发生冲突，yacc 仍然会生成解析器。如果存在归约 / 归约冲突，则优先匹配前面的语法规则，移进 / 归约冲突会优先匹配移进规则。很多书会写归约 / 归约冲突是致命错误，而移进 / 归约冲突则允许，这两者的确是存在严重程度的差别，但是在我来看，无论发生哪一种冲突都是难以容忍的，我恨不得消灭代码中所有的冲突问题。

yacc 运行时可以附带 -v 参数，标准 yacc 会生成 y.output 文件（bison 则会将输入文件名中的扩展名 .y 替换为 .output 并生成）。

这个 y.output 文件会包含所有的语法规则、解析器、所有可能的分支状态以及编译器启动信息。

那么我们实际做出一个冲突，然后观察一下 y.output 文件吧。

将代码清单 2-2 中的语法规则

```
expression
    : term
    | expression ADD term     ←将这里的 ADD
```

更改为下文所示：

```
expression
    : term
    | expression MUL term     ←更改为 MUL
```

变更后会产生 3 个移进 / 归约冲突。

```
% yacc -dv mycalc.y
conflicts: 3 shift/reduce
```

然后再看 y.output 文件。

根据 yacc 或 bison 的版本与语言设置，y.output 的输出会有微妙的区别。这里以 bison2.3 英文模式为例（为了节约纸张将空行都去掉了）。日语环境下，“Grammar”会变成“文法”等，错误信息也都会显示为日语。

总体来说，y.output 文件的前半部分看起来基本都会是下面这个样子（代码清单 2-4）。

代码清单 2-4
y.output（前半部分）

```
Terminals which are not used     ←没有使用 ADD 的警告
   ADD

State 5 conflicts: 1 shift/reduce     ←冲突信息（见下文）
State 14 conflicts: 1 shift/reduce
State 15 conflicts: 1 shift/reduce

Grammar
    0 $accept: line_list $end
    1 line_list: line
    2     | line_list line
    3 line: expression CR
    4 expression: term
    5     | expression MUL term
    6     |expression SUB term
    7 term: primary_expression
    8     | term MUL primary_expression
    9     | term DIV primary_expression
   10 primary_expression: DOUBLE_LITERAL
```

首先将 ADD 改为 MUL 后，开头会出现“ADD 没有被使用”的警告。下面会有 3 行冲突信息，此处后文会细说。再后面的“Grammar”跟着的是 mycalc.y 中定义的语法规则。

mycalc.y 中可以使用 |（竖线，表示或）书写下面这样的语法规则：

```
line_list : line
    | line_list line
```

实际上与下面这种写法的语法规则是完全一样的。

```
line_list : line
line_list : line_list line
```

y.output 文件中，会给每一行规则附上编号。

上面的规则 0，是 yacc 自动附加的规则，`$accept` 代表输入的内容，`$end` 代表输入结束，目前还不需要特别在意这个问题。

y.output 文件的后半部分会将解析器可能遇到的所有“状态”全部列举出来（代码清单 2-5）。

代码清单 2-5
y.output（后半部分）

```
state 0

    0 $accept: . line_list $end

    DOUBLE_LITERAL  shift, and go to state 1

    line_list           go to state 2
    line                go to state 3
    expression          go to state 4
    term                go to state 5
    primary_expression  go to state 6


state 1

   10 primary_expression: DOUBLE_LITERAL .

    $default  reduce using rule 10 (primary_expression)


state 2

    0 $accept: line_list . $end
    2 line_list: line_list . line

    $end            shift, and go to state 7
    DOUBLE_LITERAL  shift, and go to state 1

    line                go to state 8
    expression          go to state 4
    term                go to state 5
    primary_expression  go to state 6


state 3

    1 line_list: line .

    $default  reduce using rule 1 (line_list)


state 4

    3 line: expression . CR
```

```
    5 expression: expression . MUL term
    6           | expression . SUB term

    SUB  shift, and go to state 9
    MUL  shift, and go to state 10
    CR   shift, and go to state 11

下略
```

前文中有一个俄罗斯方块的比喻，读者通过那个比喻可能会这样理解：解析器在一个记号进入栈后，从栈的开头一个字符一个字符扫描，如果发现这个记号满足已有的某个规则，就去匹配该规则。但其实这样做会让编译器变得很慢。

现实中，yacc 在生成解析器的阶段，就已经将解析器所能遇到的所有状态都列举出来，并做成了一个解析对照表（Parse Table），表中记录了“状态 A 下某种记号进入后会转换到状态 B”这样的映射关系，y.output 的后半部分就罗列了所有可能的映射关系。听说这有点像麻将中的听牌，不过因为我不玩麻将，所以也不是很清楚。

有了这些列举出的状态，当记号进入后，就可以很容易找到在何种状态下移进，或者根据何种规则归约。那么对于之前我们故意做出的冲突，在我的环境下，y.output 开头会输出如下信息：

```
State 5 conflicts: 1 shift/reduce
State 14 conflicts: 1 shift/reduce
State 15 conflicts: 1 shift/reduce
```

可以看到 state 5 引起了冲突，我们来看一下：

```
state 5

    4 expression: term .
    8 term: term . MUL primary_expression
    9     | term . DIV primary_expression

    MUL  shift, and go to state 12
    DIV  shift, and go to state 13

    MUL       [reduce using rule 4 (expression)]
    $default  reduce using rule 4 (expression)
```

上例中，第一行的 `state 5` 即为状态的编号，1 个空行后接下来的 3 行是 y.output 前半部分中输出的语法规则（语法规则还附加了编号）。语法规则中间有 `.`，代表记号在当前规则下能被转换到哪个程度。比如 state 5 这个状态下，记

号最多被转换为 term，然后需要等待下一个记号进行归约。

再空一行，接下来记录的是当前状态下，下一个记号进入时将如何变化。具体来讲，这里当 MUL（*）记号进入后会进行移进并转换为 state 12。如果进入的是 DIV（/），则同样进行移进并转移到 state 13。

再经过 1 个空行后下一行是：

```
MUL       [reduce using rule 4 (expression)]
```

意思是当 MUL 进入后，可以按照规则 4 进行归约。这也就是移进 / 归约冲突。

yacc 默认移进优先，所以 MUL 进入后会转移到状态 12。在 y.output 中 state 12 是这样的：

```
state 12

    8 term: term MUL . primary_expression

    DOUBLE_LITERAL  shift, and go to state 1

    primary_expression  go to state 16
```

而如果是归约的情况，所要参照的规则 4 是这样的：

```
4 expression: term
```

也就是说，当记号被转换为 term 后，下一个记号 * 进入以后，yacc 会报告有冲突发生，冲突的原因是当前 term 既可以继续进行移进，也可以归约为 expression 并结束。而这正是由于我们将 ADD 修改为 MUL 后，* 的运算优先级被降低而引发的混乱。

对 y.output 阅读方法的说明就此告一段落，其实在实践中想要探明冲突原因并解决冲突，是比较有难度的。

因此即便表面上看起来都一样的编程语法，根据语法规则的书写方式不同，yacc 可能报冲突也可能不报冲突（可以参考后文讲到的 LALR(1) 语法规定）。比如本节开头所说的 C 语言中 if else 语法模糊的问题，在 Java 中就从语法规则入手回避了这个冲突。而类似这样的小问题以及相应的解决“窍门”，在自制编程语言中数不胜数，所以从现实出发，模仿已有的编程语言时，最好也要多多参考其语法规则。

2.2.6 错误处理

前面的小节中介绍了如何应对语法规则中的冲突问题，即编译器制作阶段的错误处理。而在自制编程语言时，还要考虑到使用这门语言编程的人如果犯了错误该怎么办。

在稍微旧一点的编译器书籍中，常常能读到下面这样的文字。

- 让用户自己不断的编译，既浪费 CPU 资源也浪费时间，因此编译器最好只经过一次编译就能看到大部分错误信息。
- 但是程序中的一个错误，其原因可能是由于其他错误引起的（可能会引起错误信息的雪崩现象）。一边不希望无用的错误信息出现，一边又想尽可能多地显示错误的原因，这其中的平衡点是很难掌握的。

然而时至今日，CPU 已经谈不上什么“浪费资源”了，因为在多数情况下，编译过程往往一瞬间就能结束。那么即便一次编译显示出很多错误信息，用户一般也只会看第一条（我就是这样），这样的话，“编译器最好只经过一次编译就能看到大部分错误信息”也就没什么意义了。

所以最省事的解决方法之一就是，一旦出错，立即使用 `exit()` 退出。之后章节中的 crowbar 和 Diksam 都是这样处理的。

但是对于计算器来说，是需要与用户互动，如果输错了一点程序就强制退出的话，对用户也太不友好了。

因此我们可以利用 yacc 的功能实现一个简单的错误恢复机制。

首先在 mycalc.y 的非终结符 `line` 的语法规则中，追加下面的部分。

```
line
    : expression CR
    {
        printf(">>%lf\n", $1);
    }
    | error CR
    {
        yyclearin;
        yyerrok;
    }
    ;
```

这里新出现的是 `error` 记号。`error` 记号是匹配错误的特殊记号。`error` 可以后接 `CR`（换行符），这样书写可以匹配包含了错误的所有记号以及行尾。

动作中的 yyclearin 会丢弃预读的记号，而 yyerrok 则会通知 yacc 程序已经从错误状态恢复了。

既然是有交互的工具，一般都会使用换行来分割每次的会话，每一个错误信息后加上换行符应该会显得更美观吧。

2.3 不借助工具编写计算器

至此我们使用 yacc 和 lex 制作了一个计算器，可能会有读者这样想：

- 我明明是为了弄清楚编程语言的内部机制才要自制编程语言的，但是却将 yacc 和 lex 等工具当作黑匣子使用，这样一来岂不是达不到目的了吗？
- bison 和 flex 虽然都是自由软件，但是在项目中客户和上级是不允许使用自由软件的；
- yacc/lex 或 bison/flex 的版本升级可能会使程序无法工作，这很让人讨厌。

上面每一条理由都足够充分（上级不允许使用自由软件可能有点不讲理，但是光嘴上说不讲理是没法解决这个问题的）。

因此，以下我们将不会借助 yacc/lex 来制作计算器。

2.3.1 自制词法分析器

首先是词法分析器。

操作本章的计算器时，会将换行作为分割符，把输入分割为一个个算式。跨复数行的输入是无法被解析为一个算式的，因此词法分析器中应当提供以下的函数：

```
/* 将接下来要解析的行置入词法分析器中 */
void set_line(char *line);

/* 从被置入的行中，分割记号并返回
 * 在行尾会返回 END_OF_LINE_TOKEN 这种特殊的记号
 */
void get_token(Token *token);
```

get_token() 接受的入口参数为一个 Token 结构体指针，函数中会分割出记号的信息装入 Token 结构体并返回。上面两个函数的声明以及 Token 结构体的

定义位于 token.h 文件中。

代码清单 2-6
token.h

```
#ifndef TOKEN_H_INCLUDED
#define TOKEN_H_INCLUDED

typedef enum {
   BAD_TOKEN,
   NUMBER_TOKEN,
   ADD_OPERATOR_TOKEN,
   SUB_OPERATOR_TOKEN,
   MUL_OPERATOR_TOKEN,
   DIV_OPERATOR_TOKEN,
   END_OF_LINE_TOKEN
} TokenKind;

#define MAX_TOKEN_SIZE (100)

typedef struct {
   TokenKind kind;
   double      value;
   char        str[MAX_TOKEN_SIZE];
} Token;

void set_line(char *line);
void get_token(Token *token);

#endif /* TOKEN_H_INCLUDED */
```

词法分析器的源代码如代码清单 2-7 所示。

代码清单 2-7
lexicalanalyzer.c

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "token.h"

static char *st_line;
static int st_line_pos;

typedef enum {
    INITIAL_STATUS,
    IN_INT_PART_STATUS,
    DOT_STATUS,
    IN_FRAC_PART_STATUS
} LexerStatus;

void
get_token(Token *token)
{
```

```
    int out_pos = 0;
    LexerStatus status = INITIAL_STATUS;
    char current_char;

    token->kind = BAD_TOKEN;
    while (st_line[st_line_pos] != '\0') {
        current_char = st_line[st_line_pos];
        if ((status == IN_INT_PART_STATUS || status == IN_FRAC_
            PART_STATUS)
            && !isdigit(current_char) && current_char != '.') {
            token->kind = NUMBER_TOKEN;
            sscanf(token->str, "%lf", &token->value);
            return;
        }
        if (isspace(current_char)) {
            if (current_char == '\n') {
                token->kind = END_OF_LINE_TOKEN;
                return;
            }
            st_line_pos++;
            continue;
        }

        if (out_pos >= MAX_TOKEN_SIZE-1) {
            fprintf(stderr, "token too long.\n");
            exit(1);
        }
        token->str[out_pos] = st_line[st_line_pos];
        st_line_pos++;
        out_pos++;
        token->str[out_pos] = '\0';

        if (current_char == '+') {
            token->kind = ADD_OPERATOR_TOKEN;
            return;
        } else if (current_char == '-') {
            token->kind = SUB_OPERATOR_TOKEN;
            return;
        } else if (current_char == '*') {
            token->kind = MUL_OPERATOR_TOKEN;
            return;
        } else if (current_char == '/') {
            token->kind = DIV_OPERATOR_TOKEN;
            return;
        } else if (isdigit(current_char)) {
            if (status == INITIAL_STATUS) {
                status = IN_INT_PART_STATUS;
```

```
            } else if (status == DOT_STATUS) {
                status = IN_FRAC_PART_STATUS;
            }
        } else if (current_char == '.') {
            if (status == IN_INT_PART_STATUS) {
                status = DOT_STATUS;
            } else {
                fprintf(stderr, "syntax error.\n");
                exit(1);
            }
        } else {
            fprintf(stderr, "bad character(%c)\n", current_char);
            exit(1);
        }
    }
}

void
set_line(char *line)
{
    st_line = line;
    st_line_pos = 0;
}

/* 下面是测试驱动代码 */
void
parse_line(char *buf)
{
    Token token;

    set_line(buf);

    for (;;) {
        get_token(&token);
        if (token.kind == END_OF_LINE_TOKEN) {
            break;
        } else {
               printf("kind..%d, str..%s\n", token.kind, token.str);
        }
    }
}

int
main(int argc, char **argv)
{
    char buf[1024];

```

```
112:     while (fgets(buf, 1024, stdin) != NULL) {
113:         parse_line(buf);
114:     }
115:
116:     return 0;
117: }
```

这个词法分析器的运行机制为，每传入一行字符串，就会调用一次 get_token() 并返回分割好的记号。由于词法分析器需要记忆 set_line() 传入的行，以及该行已经解析到的位置，所以设置了静态变量 st_line 与 st_line_pos（第 6 ～ 7 行）*。

* 按本书所采用的命名规范，文件内的 static 变量需要附带前缀 st_。请参考 3.2.1 节。

set_line() 函数，只是单纯设置了 st_line 与 st_line_pos 的值。

get_token() 则负责将记号实际分割出来，即词法分析器的核心部分。

第 24 行开始的 while 语句，会逐一按照字符扫描 st_line。

记号中的 +、-、*、/ 四则运算符只占一个字符长度，因此一旦扫描到了，立即返回就可以了。

数值部分要稍微复杂一些，因为数值由多个字符构成。鉴于我们采用的是 while 语句逐字符扫描这种方法，当前扫描到的字符很有可能只是一个数值的一部分，所以必须想个办法将符合数值特征的值暂存起来。为了暂存数值，我们采用一个枚举类型 LexerStatus* 的全局变量 status（第 20 行）。

* 数值可以分为整数部分、小数点和小数部分。使用 lex 时，用一个正则表达式描述其特征，之后全部交给 lex 处理。而在这里我们就只能自力更生了。

首先，status 的初始状态是 INITIAL_STATUS。当遇到 0 ～ 9 的数字时，这些数字会被放入整数部分（此时状态为 IN_INT_PART_STATUS）中（第 64 行）。一旦遇到小数点 .，status 会由 IN_INT_PART_STATUS 切换为 DOT_STATUS（第 70 行），DOT_STATUS 再遇到数字会切换到小数状态（IN_FRAC_PART_STATUS，第 66 行）。在 IN_INT_PART_STATUS 或 IN_FRAC_PART_STATUS 的状态下，如果再无数字或小数点出现，则结束，接受数值并 return。

按上面的处理，词法分析器会完全排除 .5 或 2..3 这样的输入。而从第 32 行开始的处理，除换行以外的空白符号全部会被跳过。

由于是用于计算器的词法分析器，因此除四则运算符与数值外，没有其他要处理的对象了，但如果考虑到将其扩展并可以支持编程语言的话，最好提前想到以下几个要点。

1. 数值与标识符（如变量名等）可以按照上例的方法通过管理一个当前状态将其解析出来，比如自增运算符就可以设置一个类似IN_INCREMENT_OPERATOR的状态，但这样一来程序会变得冗长。因此对于运算符来说，可能为其准备一个字符串数组会更好。比如做一个下面这样的数组：

```
static char *st_operator_str[] = {
    "++",
    "--",
    "+",
    "-",
    （以下省略）
};
```

   当前读入的记号可以与这个数组中的元素做前向匹配，从而判别记号的种类。指针部分同样需要比特征对象再多读入一个字符用以判别（比如输入i+2，就需要将2也读入看看有没有是i++的可能性）。做判别时，像上例这样将长的运算符放置在数组前面会比较省事。关于额外读入的一个字符具体应该如何处理，稍后会介绍。

   另外，像if、while这些保留字，比较简单的做法是先将其判别为标识符，之后再去对照表中查找有没有相应的保留字。

2. 本次的计算器是以行为单位的，st_line会保存一行中的所有信息，但在当下的编程语言中，换行一般和空白字符是等效的，因此不应该以行为单位处理，而是从文件中逐字符（使用getc()等函数）读入解析会更好。

   那么，上例中用while语句逐字符读取的地方就需要替换为用getc()等函数来读取，比如输入123.4+2时，判别数值是否结束的时机是读入+时。

   上例的词法分析器是通过st_line_pos的自增（第46行st_line_pos++）来实现的。如果直接从文件逐字符读入，C语言中就需要使用ungetc()等从读入的字符回退，从而产生1个字符的备份，达到预先读入下一字符的效果。

补充知识　保留字（关键字）

在 C 语言中，if 与 while 都是保留字，保留字无法再作为变量名使用【C 规范中一般不称“保留字”（reserved word），而称为“关键字”（keyword），但是关键字的指代范围太广，所以还是称保留字更加准确】。

C 语言中的保留字是由词法分析器以特殊的标识符方式处理的。保留字的区分以标识符为单位，比如 if 不能作为变量名但 ifa 就可以。

对于习惯了 C 语言的人来说，这都是理所当然的事情，但站在其他语言的角度看却未必如此。

比如在我小时候折腾过一门叫 BASIC 的编程语言 *，可以这样写：

```
IFA=10THEN...
```

* 这里的例子仅限于 BASIC 的旧版本，在 N88-BASIC 中 IFA 的写法也是不允许的。

会解析为：

```
IF A = 10 THEN ...
```

向杂志投稿的程序中，注释（英语为 remark）都写成了下面这样：

```
REMARK 这里是注释
```

BASIC 中的注释只需要写 `REM` 语句，`REM` 之后都会被作为注释处理，因此即便写成 `REMARK` 也是可以的。

BASIC 是从 FORTRAN 的基础上发展起来的，以前 FORTRAN 中空白字符没有任何意义。GOTO 可以写成 `GO TO`，也可以写成 `G OTO`，而在写循环的时候，下例等于写了一个 1 到 5 的循环。

```
    DO 10 I=1,5
    处理
10  CONTINUE
如果一不小心将逗号输入成句号，写成下面这样：
    DO 10 I=1.5
```

由于 FORTRAN 中空白没有意义，而且上例中也无需声明变量（D 开始的变量默认解析成实数型变量），所以最后会变成 `DO10I=1.5` 这样的赋值语句。有传闻 * 说就是这样一个 BUG 最终导致 NASA 的火箭失控爆炸，当然这多半是谣传了。

* 文章的标题是 “Fortran story – the real scoop”，是当时在 NASA 的 Fred Webb 向新闻组 alt.folklore.computers 的一篇投稿，有兴趣的朋友可以去搜索一下。

另外在 C# 中还有**上下文关键字**(context keyword)，是指一些在特殊的区域内才对编译器有特殊意义的关键字（比如定义属性时使用 get 等）。内容关键字并不等同于保留字，在普通的变量名中可以使用。保留字与关键字严格讲有不同的意义，但本书中没有特别区分。

补充知识　避免重复包含

在代码清单 2-6 中，开头和结尾处有这样的语句：

```
#ifndef TOKEN_H_INCLUDED
#define TOKEN_H_INCLUDED
（中间省略）
#endif /* TOKEN_H_INCLUDED */
```

这是为了防止 token.h 多次用 `#include` 包含引起多重定义错误而采用的技巧。

头文件经常会用到其他头文件中定义的类型或宏。比如在 a.h 中定义的类型在 b.h 中使用的话，在 b.h 的开头处书写 `#include "a.h"` 就可以了。如果不这样做，程序用 `#include` 包含 b.h 时，必须同时书写 `#include a.h` 与 `#include b.h`，还会弄得代码到处都是长串 `#include`，之后如果依赖关系发生改变的话修改起来非常麻烦。

但仅仅在头文件的起始处用 `#include` 包含 a.h，如果多个头文件都这样书写，会



报出类型或宏的重复定义错误。因此采用上面的小技巧，一旦 token.h 用 #include 包含后会定义 TOKEN_H_INCLUDED，根据开头的 #ifndef 语句，该头文件将被忽略，也就避免了产生多重定义的错误。

下面的两点是编写 C 头文件的经验之谈，本书中涉及的代码都默认遵循这两点：

1. **所有的头文件都必须用 #include 包含自己所依赖的其他所有头文件，最终让代码中只需一次 #include；**
2. **所有的头文件都必须加入上文的技巧，防止出现多重定义错误。**

2.3.2 自制语法分析器

接下来终于要开始做语法分析器了。

在我看来，只要是有一定编程经验的程序员，即使没有自制编程语言的背景，都可以大致想明白词法分析器的运行机制。但换成语法分析器，可能很多人就有点摸不着头脑了。有些人可能会想，总之先只考虑计算器程序，将运算符优先级最低的 + 与 - 分割出来，然后再处理 * 和 /……这样的思路基本是正确的。但是按这样的思路实际操作时会发现，用来保存分割字符串的空间可能还有其他用途，而加入括号的处理也很难。

对于上面的问题，与其自己想破脑袋，不如借鉴一下前人的智慧。因此我们将使用一种叫**递归下降分析**的方法来编写语法分析器。

yacc 版的计算器曾使用下面的语法规则：

```
expression  /* 表达式的规则 */
        : term  /* 表达式 */
        | expression ADD term  /* 或 表达式 + 表达式 */
        | expression SUB term  /* 或 表达式 - 表达式 */
        ;
term  /* 表达式的规则 */
        : primary_expression  /* 一元表达式 */
        | term MUL primary_expression  /* 或 表达式 * 表达式 */
        | term DIV primary_expression  /* 或 表达式 / 表达式 */
        ;
primary_expression  /* 一元表达式的规则 */
        : DOUBLE_LITERAL  /* 实数的字面常量 */
        ;
```

这些语法规则可以用图 2-5 这样的**语法图**（syntax graph 或 syntax diagram）来表示。

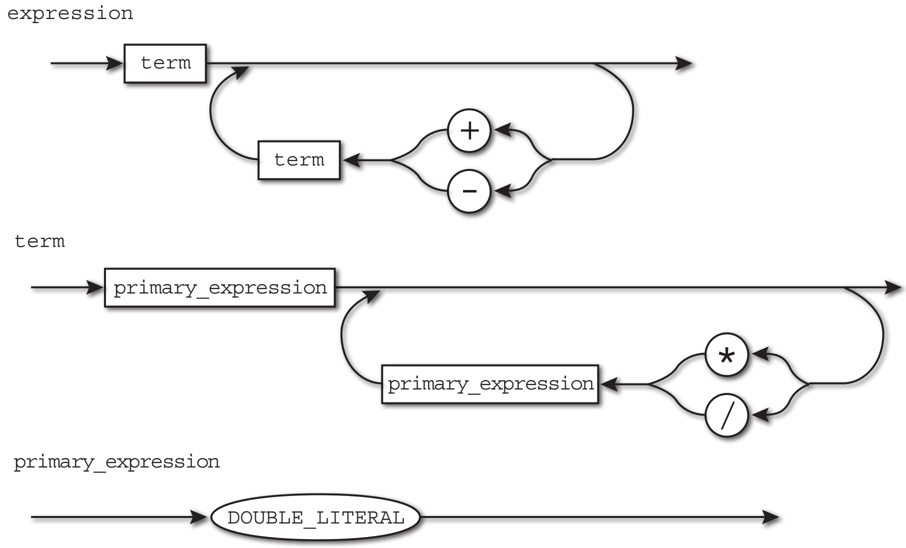


图 2-5
计算器的语法图

语法图的表示方法应该一看就能明白，比如项目（term）的语法图代表最初进入一元表达式（primary_expression），一元表达式可以直接结束，也可以继续进行 * 或 / 运算，然后又有一个一元表达式进入，重复这一流程。作为语法构成规则的说明，语法图要比 BNF 更容易理解吧。

本书的语法图例中，非终结符用长方形表示，终结符（记号）用椭圆形表示。

正如语法图所示，递归下降分析法读入记号，然后执行语法分析。

比如解析一个项目（term）的函数 parse_term()，如代码清单 2-8 所示，按照语法图所示流程工作。

代码清单 2-8
parser.c（节选）

```
    /* primary expression 的解析函数 */
    v1 = parse_primary_expression();
    for (;;) {
        my_get_token(&token);
        /* 循环扫描 “*”、“/” 以外的字符 */
        if (token.kind != MUL_OPERATOR_TOKEN
            && token.kind != DIV_OPERATOR_TOKEN) {
            /* 将记号 Token 退回 */
            unget_token(&token);
            break;
        }
        /* primary expression 的解析函数 */
        v2 = parse_primary_expression();
        if (token.kind == MUL_OPERATOR_TOKEN) {
            v1 *= v2;
```

```
62:         } else if (token.kind == DIV_OPERATOR_TOKEN) {
63:             v1 /= v2;
64:         }
65:     }
66:     return v1;
```

如同语法图中最开始的 primary_expression 进入一样，第 51 行的 parse_primary_expression() 会被调用。递归下降分析法中，一个非终结符总对应一个处理函数，语法图里出现非终结符就代表这个函数被调用。因此在第 52 行下面的 for 语句会构成一个无限循环，如果 *（MUL_OPERATOR）与 /（DIV_OPERATOR）进入，循环会持续进行（其他字符进入则通过第 57 行的 break 跳出）。而第 59 行第二次调用 parse_primary_expression()，与语法图中 * 和 / 右边的 primary expression 相对应。

比如遇到语句 1 * 2 + 3，第 51 行的 parse_primary_expression() 将 1 读入，第 53 行 my_get_token() 将 * 读入，接下来第 59 行的 parse_primary_expression() 将 2 读入。之后的运算符根据种类不同分别执行乘法（第 61 行）或除法（第 63 行）。

至此已经计算完毕 1 * 2，然后第 53 行的 my_get_token() 读入的记号是 +。+ 之后再没有 term 进入，用 break 从循环跳出。但由于此时已经将 + 读进来了，因此还需要用第 56 行的 unget_token() 将这个记号退回。parser.c 没有直接使用 lexicalanalyzer.c 中写好的 get_token()，而使用了 my_get_token()，my_get_token() 会对 1 个记号开辟环形缓冲区（Ring Buffer）（代码清单 2-9 第 7 行的静态变量 st_look_ahead_token 是全部缓冲），可以借用环形缓冲区将最后读进来的 1 个记号用 unget_token() 退回。这里被退回的 +，会重新通过 parse_expression() 第 78 行的 my_get_token() 再次读入。

完整代码如代码清单 2-9 所示。

根据语法图的流程可以看到，当命中非终结符时，会通过递归的方式调用其下级函数，因此这种解析器称为递归下降解析器。

这个程序作为一个带有运算优先级功能的计算器来说，代码是不是出乎意料地简单呢。那么请尝试对各种不同的算式进行真机模拟，用 debug 追踪或者 printf() 实际调试一下吧。

代码清单 2-9
parser.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "token.h"

#define LINE_BUF_SIZE (1024)

static Token st_look_ahead_token;
static int st_look_ahead_token_exists;

static void
my_get_token(Token *token)
{
    if (st_look_ahead_token_exists) {
        *token = st_look_ahead_token;
        st_look_ahead_token_exists = 0;
    } else {
        get_token(token);
    }
}

static void
unget_token(Token *token)
{
    st_look_ahead_token = *token;
    st_look_ahead_token_exists = 1;
}

double parse_expression(void);

static double
parse_primary_expression()
{
    Token token;

    my_get_token(&token);
    if (token.kind == NUMBER_TOKEN) {
        return token.value;
    }
    fprintf(stderr, "syntax error.\n");
    exit(1);
    return 0.0; /* make compiler happy */
}

static double
parse_term()
{
    double v1;
```

```
    double v2;
    Token token;

    v1 = parse_primary_expression();
    for (;;) {
        my_get_token(&token);
        if (token.kind != MUL_OPERATOR_TOKEN
            && token.kind != DIV_OPERATOR_TOKEN) {
            unget_token(&token);
            break;
        }
        v2 = parse_primary_expression();
        if (token.kind == MUL_OPERATOR_TOKEN) {
            v1 *= v2;
        } else if (token.kind == DIV_OPERATOR_TOKEN) {
            v1 /= v2;
        }
    }
    return v1;
}

double
parse_expression()
{
    double v1;
    double v2;
    Token token;

    v1 = parse_term();
    for (;;) {
        my_get_token(&token);
        if (token.kind != ADD_OPERATOR_TOKEN
            && token.kind != SUB_OPERATOR_TOKEN) {
            unget_token(&token);
            break;
        }
        v2 = parse_term();
        if (token.kind == ADD_OPERATOR_TOKEN) {
            v1 += v2;
        } else if (token.kind == SUB_OPERATOR_TOKEN) {
            v1 -= v2;
        } else {
            unget_token(&token);
        }
    }
    return v1;
}
```

```c

double
parse_line(void)
{
    double value;

    st_look_ahead_token_exists = 0;
    value = parse_expression();

    return value;
}

int
main(int argc, char **argv)
{
    char line[LINE_BUF_SIZE];
    double value;

    while (fgets(line, LINE_BUF_SIZE, stdin) != NULL) {
        set_line(line);
        value = parse_line();
        printf(">>%f\n", value);
    }

    return 0;
}
```

补充知识　预读记号的处理

本书中采用的递归下降解析法，会预先读入一个记号，一旦发现预读的记号是不需要的，则通过 unget_token() 将记号“退回”。

换一种思路，其实也可以考虑“始终保持预读一个记号”的方法。按照这种思路，代码清单 2-9 可以改写成代码清单 2-10 这样：

代码清单 2-10
parser.c（始终保持预读版）

```c
/* token 变量已经放入了下一个记号 */
parse_primary_expression();
for (;;) {
    /* 这里无需再读入记号 */
    if (token.kind != MUL_OPERATOR_TOKEN
        && token.kind != DIV_OPERATOR_TOKEN) {
        /* 不需要退回处理 */
        break;
    }
```

```
        /* token.kind 之后还会使用，所以将其备份
         * 而parse_primary_expression() 也就可以读入新的记号
         */
        kind = token.kind;
        my_get_token(&token);
        v2 = parse_primary_expression();
        if (kind == MUL_OPERATOR_TOKEN) {
            v1 *= v2;
        } else if (kind == DIV_OPERATOR_TOKEN) {
            v1 /= v2;
        }
    }
```

比较这两种实现方式，会发现两者的实质基本上是一样的。很多编译器入门书籍中列举的实例代码，和本书中的例子相差无几。

不过这里还是会有个人偏好，就我而言，更喜欢“边读入边退回”的方法。在“始终保持预读”的方法中，变量 token 是一个全局变量，代码中相隔很远的地方也会操作其变量值，追踪数据变化会比较麻烦。

2.4 少许理论知识——LL(1) 与 LALR(1)

2.3.2 节中手写的解析器会对记号进行预读，并按照语法图的流程读入所有记号。这种类型的解析器叫作**LL(1) 解析器**。LL(1) 解析器所能解析的语法，叫作 LL(1) 语法。

采用 LL(1) 语法，当然能制作出对应的编程语言来。比如 Pascal 的语法就是 LL(1)。

但是看了代码清单 2-9 就能明白，LL(1) 解析器在语法上需要非终结符与解析器内部的函数一一对应。也就是说，只看第一个进入的记号，还无法判断需不需要继续往下读取，也不能知道当前非终结符究竟是什么。

比如在 Pascal 中，goto 语句使用的标签只能是数字，这样限制的原因是，如果像 C 语言一样允许英文字母作为标识符的话，读入第一个记号时，就没有办法区分这个记号究竟是赋值语句的一部分，还是标签语句的一部分。因为无论赋值语句还是标签语句，开始的标识符是一样的。由此可知，LL(1) 语法所做出的解析器都比较简单，语法能表达的范围比较狭窄。

那么，在把计算器的 BNF 改写为语法图的过程中，一些敏锐的读者可能已经有了这样的疑问：

不管是用 BNF 还是语法图，都应该只是表面上有区别，语法实现部分应该是一样的啊。但你写的代码怎么连算法都不一样？我有种上当了的感觉。

实际上确有此事，在把 BNF 置换为图 2-5 所示的语法图时，我运用了一个小手法。在 BNF 中语法规则是这样的：

```
expression /* 表达式的规则 */
        | expression ADD term /* 或 表达式 + 项目 */
```

而在实现递归下降分析时，如果仍然按这个规则在 `parse_expression()` 刚开始就调用 `parse_expression()`，会造成死循环，一个记号也读不了。

BNF 这样的语法称为**左递归**，原封照搬左递归的语法规则，是无法实现递归下降分析的。

所以 yacc 生成的解析器称为 **LALR(1) 解析器**，这种解析器能解析的语法称为 **LALR(1) 语法**。LALR(1) 解析器是 LR 解析器的一种。

LL(1) 的第一个 L，代表记号从程序源代码的最左边开始读入。第二个 L 则代表**最左推导**（Leftmost derivation），即读入的记号从左端开始置换为分析树。而与此相对的 LR 解析器，从左端开始读入记号与 LL(1) 解析器一致，但是发生归约时（参看 2.2.3 节图 2-3），记号从右边开始归约，这称为**最右推导**（Rightmost derivation），即 LR 解析器中 R 字母的意思。

递归下降分析会按自上而下的顺序生成分析树，所以称作递归“下降”解析器或递归“向下”解析器。而 LR 解析器则是按照自下而上的顺序，所以也称为“自底向上”解析器。

此外，LL(1)、LALR(1) 等词汇中的 (1)，代表的是解析时所需前瞻符号（lookahead symbol），即记号的数量。

LALR(1) 开头的 LA 两个字母，是 Look Ahead 的缩写，可以通过预读一个记号判明语法规则中所包含的状态并生成语法分析表。LALR 也是由此得名的。

本章中实际制作的计算器是采用 LL(1) 语法作为解析器的，因为比较简单，所以适合手写。如果是 LALR(1) 等 LR 语法的话，则更适合用 yacc 等工具自动生成（这话可能已经说了太多遍了）。不过，最近像 ANTLR、JavaCC 等一些采用 LL(k)，即预读任意个记号的 LL 解析器也开始普及起来。

补充知识　Pascal/C 中的语法处理诀窍

前面提到 Pascal 采用的是 LL(1) 语法，但是在 Pascal 中，同时存在赋值语句和过程调用（C 语言中是函数调用）。按照之前的介绍，这两者都由同一类标识符开始的，LL(1) 解析器似乎无法区分。

在这个问题上，Pascal 并没有从一开始就强行将其区分，而是逆转思路，引入了一个同时代表“赋值语句或过程调用”的非终结符，然后在下一个记号读入后再将其分开。这样不用更改 Pascal 语法设计，仅仅变化一下语法规则就解决了问题。

在 C 语言中，如果是通过 `typedef` 命名的一些类型，其标识符 yacc（LALR(1) 解析器）是无法解析的。比如 C 语言中可以简单地声明为：

```
Hoge *hoge_p = NULL;
```

其中的星号究竟是乘法运算符还是指针符号，单看 Hoge 这个标识符很难直观得出结论。

对此，C 语言用了一个小诀窍，即在标识符作为类型名被声明的时候，会由语法分析器通知词法分析器：此后凡遇到这个标识符，不要将其作为标识符，而作为类型名返回。

通过很多类似的诀窍，终于可以让 LL(1)/LALR(1) 解析器解析 Pascal/C 语言了。C 语言图书 K&R* 的附录中，就记录了 BNF 要经过一些修正才可以输入 yacc 的内容。

* 即被称为“C 语言圣经”的 *The C Programming Language* 一书，作者名缩写为 K&R[2]。中文版为《C 程序设计语言（第 2 版・新版）》，机械工业出版社于 2004 年出版。

2.5　习题：扩展计算器

2.5.1　让计算器支持括号

如果要说普通计算器有什么不方便的地方，不能直接输入括号进行计算就是其中之一。因此为了让 `mycalc` 支持括号，我做了一些修改。

因为使用的是 yacc/lex，所以首先在 lex 中增加（和）两个记号。

```
（前略）
"+"             return ADD;
"-"             return SUB;
"*"             return MUL;
"/"             return DIV;
"("             return LP;    ←新增
")"             return RP;    ←新增
"\n"            return CR;
（后略）
```

LP、RP 分别是 left paren、right paren 的缩写。

然后将 primary_expression 的语法规则替换为下面这样：

```
primary_expression
    : DOUBLE_LITERAL
    | LP expression RP
    {
        $$ = $2;
    }
    ;
```

一看就能明白，意思是被（ ）包裹的 expression 还是一个 primary_expression。

不过仅这两处修改还不能让 mycalc 支持括号。

使用递归下降分析法制作语法分析器的话，primary_expression 的语法图需更改为图 2-6 那样。

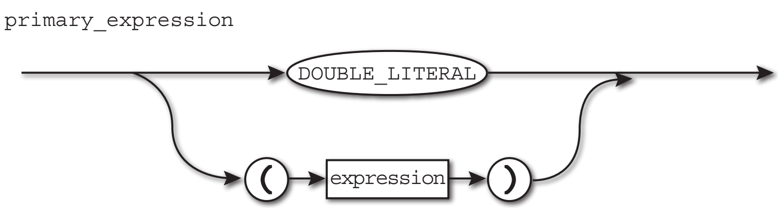


图 2-6 在语法图中引入括号

这表示用括号将 expression 包裹的部分，整体将会作为 primary_expression 来处理。

那么按这个思路重新编写 parser.c 如下所示。

```
static double
parse_primary_expression()
{
    Token token;
    double value;

    my_get_token(&token);
    if (token.kind == NUMBER_TOKEN) {
        return token.value;
    } else if (token.kind == LEFT_PAREN_TOKEN) {
        value = parse_expression();
        my_get_token(&token);
        if (token.kind != RIGHT_PAREN_TOKEN) {
            fprintf(stderr, "missing ')' error.\n");
            exit(1);
        }
        return value;
```

```
    } else {
        unget_token(&token);
        return 0.0; /* make compiler happy */
    }
}
```

将语法图直接转换为代码应该不是很难，只要按图中的思路去做即可。如果进入的不是 DOUBLE_LITERAL 而是 (，则把括号中的部分作为一个 expression 去解析就可以了。此外，如果 expression 解析完毕后没有找到标记结束的右括号)，则需要报错。

2.5.2 让计算器支持负数

其实目前做出来的计算器，还无法支持负数。因为在定义数值时用的正则表达式是 [1-9][0-9]* 或 [0-9]*\.[0-9]*，根本没有把负数作为一种数值考虑进来。

那么，如果我们想修改计算器让其支持负数，要怎么办呢？

可能有人会想，只要在词法分析器中将 -5 这样的输入也作为 DOUBLE_LITERAL 来处理不就行了吗？按这种思路，3-5 这样的输入会被解析成 3 和 -5 两个记号（请参考 2.1 节中的补充知识）。

因此，如果不想将负数作为记号处理，就应该在语法分析器中想办法。

如果用 yacc 的话，我们可能首先会想到这样做：

```
primary_expression
    : DOUBLE_LITERAL
    | SUB DOUBLE_LITERAL   ←DOUBLE_LITERAL 之前带“-”的部分也解析为表达式
    {
        $$ = -$2;
    }
    （下面省略）
```

确实，用这种方法可以给定值的实数加上负号 -。

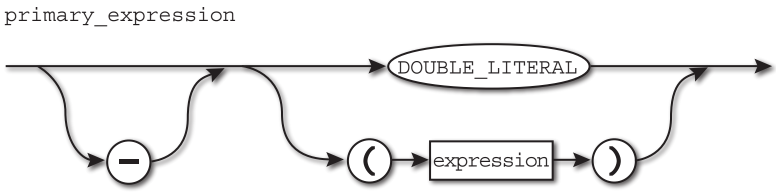
但是，用这种方法给 -(3 * 2) 这样带括号的算式再加上负号是办不到的（有人可能觉得办不到也无所谓，请允许我吹毛求疵一下）。为了再支持这种括号的处理，还需要这样修改：

```
primary_expression
    : DOUBLE_LITERAL
    | SUB primary_expression
    {
```

```
        $$ = -$2;
    }
    （下面省略）
```

那么在递归下降分析法中，可以允许负号的语法图如图 2-7 所示（这个语法图还包含了对括号的支持）。

图 2-7
包含负号的语法图

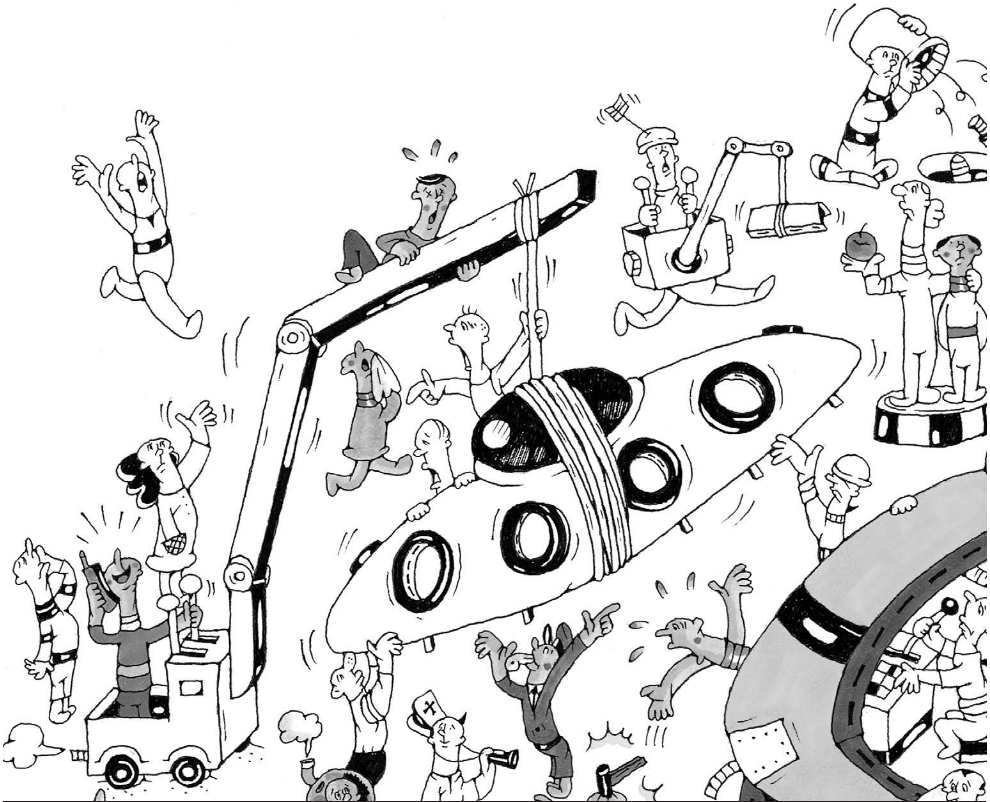


将其转换为代码，如下所示。

```
static double
parse_primary_expression()
{
    Token token;
    double value = 0.0;
    int minus_flag = 0;

    my_get_token(&token);
    if (token.kind == SUB_OPERATOR_TOKEN) {
        minus_flag = 1;
    } else {
        unget_token(&token);
    }

    my_get_token(&token);
    if (token.kind == NUMBER_TOKEN) {
        value = token.value;
    } else if (token.kind == LEFT_PAREN_TOKEN) {
        value = parse_expression();
        my_get_token(&token);
        if (token.kind != RIGHT_PAREN_TOKEN) {
            fprintf(stderr, "missing ')' error.\n");
            exit(1);
        }
    } else {
        unget_token(&token);
    }
    if (minus_flag) {
        value = -value;
    }
    return value;
}
```



第 3 章

制作无类型语言 crowbar

3.1 制作 crowbar ver.0.1 语言的基础部分

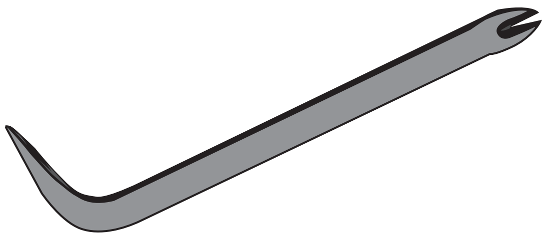
本书首先制作一门无变量类型的语言。像 Perl、Ruby、Python、PHP 这些近些年火起来的脚本语言，基本都没有变量类型。我们把将要制作的语言命名为 crowbar。

本章首先对 crowbar 的初始版本（ver.0.1）进行简要说明。

3.1.1 crowbar 是什么

crowbar 不是那种如果找到有四片叶子就会有好运降临的植物（那叫三叶草），而是如图 3-1 这样形状的工具。

图 3-1
名为 crowbar 的工具



之所以起名叫 crowbar，主要是因为这次要做的语言会生成分析树并执行。单就这点来说是与 Perl 比较接近的。有句话是怎么说来着，对了，就是那句经常能从新闻里听到的：

撬棍状的物体[①]

于是我就以 crowbar 命名了。喂，别向我扔石头啊。

如前文所述，crowbar 的语法应当照顾本书读者的习惯，所以沿袭了 C 语言的语法。

首先将初版的 crowbar 命名为 crowbar book_ver.0.1，示例代码如代码清单 3-1 所示。

---

① 当刑事案件发生时，如果物证尚不充分，警方在新闻发表会上描述犯罪所使用的道具时会经常用“撬棍状的物体”来形容，这一说法对于日本人来说是耳熟能详的。由于 Perl 与撬棍在日语中发音很接近，所以作者用 crowbar 命名其实是一语双关的小幽默。——译者注

代码清单 3-1
fizzbuzz_0_1.crb

```
for (i = 1; i < = 100; i = i + 1) {
    if (i % 15 == 0) {
        print("FizzBuzz\n");
    } elsif (i % 3 == 0) {
        print("Fizz\n");
    } elsif (i % 5 == 0) {
        print("Buzz\n");
    } else {
        print("" + i + "\n");
     }
}
```

与代码清单 1-1 不同的是，由于自增运算符 ++ 尚未实现，所以写成了 `i = i + 1`。

这个版本的 crowbar 还没有实现一门编程语言应当具备的所有基本功能（可能有读者会说，就这样也敢与 Perl 相提并论呀），当前版本所实现的功能，会在以后的章节中加以说明。

3.1.2 程序的结构

crowbar 与 Perl 一样，支持在**顶层结构**书写代码。所谓的顶部结构，即函数或类的外侧。

C 语言中，在函数的外面可以定义变量却不能书写执行语句，因此即便只写一句“hello, world”，也需要 `main()` 函数。Java 就更悲惨了，必须写长长的一串 `public class HelloWorld` 还有 `public static void main(String[] args)` 这种外行人看来像咒语一样的东西。如果仅仅想写几行简单的脚本，这实在很麻烦，而对于初学者来说也增加了学习的难度。

在 crowbar 中，如果想写一个显示“hello, world”的程序，只需简单地写成下面这样就可以了。

```
print("hello, world\n");
```

无需再包裹函数或者类。

函数的定义，需要使用保留字 `function`，按如下方式书写：

```
# 显示将 a 与 b 相加的值，并且作为返回值返回的函数
function hoge(a, b) {
    c = a + b;
```

```
    print("a+b.." + c + "\n");

    return c;
}
```

函数定义在程序中可以写在任意位置。程序执行时，首先将顶层结构中的语句从上往下顺序执行，函数定义部分会被跳过。直至函数被调用时，才执行该函数内的语句。

函数如果不存在 `return` 语句，将返回特殊的常量 `null`。

3.1.3　数据类型

可以使用的数据类型如下所示。

- 布尔型。值可以为 `true` 或 `false`。
- 整数型。其实就是 crowbar 底层运行环境的 C 语言的 `int` 型。
- 实数型。即 crowbar 底层运行环境的 C 语言的 `double` 型。当整数型与实数型混合运算时，整数型将被扩充为实数型。
- 字符串型。可以通过 + 运算符连接。另外，如果字符串在左侧数值在右侧，用 + 连接的话，右侧将被转换为字符串型。

例如：

```
print("10 + 5.." + (10 + 5));      ←将显示10 + 5..15
```

- 原生指针型（Native Pointer）。请读者不要根据名字将其想象成那种可以直接访问内存的邪恶指针，crowbar 的原生指针型类似于 C 语言的 `FILE*`，是用于在 crowbar 内部移动跳转的类型。详细请参考 3.1.7 节。

在 book_ver.0.1 中，不存在数组、关联数组（associative array）、类、对象等类型。

3.1.4　变量

crowbar 与 Perl、Ruby 等相同，都是静态无类型（即变量无需声明类型）语言。

crowbar 无需变量声明，赋初始值时就包含了声明过程（和 Ruby 非常类似）。

如果直接引用一个还没有赋值的变量则会报错。

变量的命名规则与 C 基本一样，必须以字母开头，第二个字符开始可以使用字母数字，也支持下划线。与 Perl 等不同的是，变量开头无需书写 $ 符号。

函数内首次进行赋值的变量会作为函数的局部变量，局部变量的生命周期及作用域仅限于当前函数内部。C 语言等还可以在函数中用 {} 再开辟一个块（Block），并在块内有更小作用域的局部变量，crowbar 则不支持这种特性。

变量是在赋值语句执行时进行声明的，如下例所示：

```
if(a == 10) {
    b = 10;
}
print("b.." + b);
```

a 只有为 10 的时候 b 才被声明，print 语句可以正常显示。如果 a 不为 10 则会报出未定义变量的错误。

在顶层结构中赋值的变量会成为全局变量。函数中引用全局变量时，需要用 global 语句进行声明。

global 语句可以按以下的方式使用：

```
global 变量名 , 变量名 , …;
```

比如函数内用 global a; 声明之后，在该函数内就可以引用全局变量 a（如果全局变量 a 不存在则会报运行错误）。

比如运行代码清单 3-2，运行结果如下所示：

```
a..30
a..20
```

运行结果第 1 行的 a..30 是代码清单 3-2 第 10 行的 print 输出结果，因此这里显示的是 func2() 中被赋值的局部变量 a 的值。

第 2 行的 a..20 则是第 15 行的 print 结果，显示的是全局变量 a 的值。

因为有了 global 语句，所以第 5 行赋值的是全局变量 a 的引用，而第 9 行只引用了局部变量，因此即使对其赋值也不会对全局变量产生影响。

代码清单 3-2
global.crb

```
1: a = 10;       ←定义全局变量 a 的声明
2:
3: function func() {
4:     global a;
5:     a = 20;     ←这里的 a 是全局变量
6: }
```

```
function func2() {
    a = 30;   ←这里的 a 是局部变量
    print("a.." + a + "\n");
}

func();
func2();
print("a.." + a + "\n");
```

那么，为什么一定要使用 `global` 语句声明后才可以引用全局变量呢？这样的设计有以下两个原因。

- 如果没有任何约束就可以直接引用全局变量，那么编写函数时就必须随时掌握所有全局变量的情况，而对于强调高内聚性的函数来说，这种设计会产生致命的错误。
- 全局变量的使用频率并不高，因此设置这样一点障碍对编写程序不会产生太大影响。

话虽如此，在使用 `STDIN`（标准输入的文件指针）这样的全局变量时也必须声明，还是多少有些不方便的。

补充知识　初次赋值兼做变量声明的理由

如上文所述，crowbar 会在变量初次赋值时兼做变量声明，即如果直接使用没有赋值的变量会报错。

比如在 Perl 中，默认情况下，即使没有赋值的变量仍然可以使用。此时该变量值会根据上下文自动转换。像下面这样书写的话：

```
print 123 * $a; #对未赋值的变量$a进行乘法运算
```

运行结果为 0，因为未赋值的变量 $a 的值被自动转换为 0 了。

但是这样的设计容易因为变量名输入有误而引起 BUG*。因此在 crowbar 的设计中，只能使用进行过初次赋值的变量。

* 上面的语句在 Perl 中如果加上 -w 参数并运行的话会出现警告。

还需要注意的是，crowbar 在执行变量的赋值语句时才会被声明，而 Ruby 只要书写了赋值语句就完成了变量声明，即赋值语句的执行不是必须的。因此，像下面这样：

```
x = x; #这个例子中，赋值语句执行前，x也可以使用
```

或

```
if false
  a = 1
end

print a; #赋值语句没有执行，也可以使用a。
```

这些程序在 Ruby 中都是合法的。关于这样设计的理由，Ruby 的作者松本行弘先生做了如下说明（请参考 ruby-list 邮件列表的 No.33798）：

全局变量的作用域应当通过静态方式决定，也就是说，在赋值语句开始执行才检查变量是否存在，这样的设计并不好。
因为动态的变量作用域用户理解起来有难度，同时也失去了一次编程语言中为数不多的可以进行性能优化的机会。

关于这一点我是持同意态度的 *，那为什么 crowbar 中没有这样去做呢？理由其实很简单，只是想要偷懒一下而已。

* 不过我还是有些介意，Ruby 中连类或方法的定义都是动态可执行的，为什么偏偏变量的定义要做成静态的呢。

补充说明　各种语言的全局变量处理

下面来看一看其他语言中全局变量的处理方法。

Perl：变量默认是全局的，只有加上 `local` 或 `my` 等定义后才会变成局部变量。
Ruby：用 $ 开头的变量是全局变量。
PHP：与 crowbar 一样，函数内要引用全局变量的话，用 `global` 语句定义。

一般来说，程序中应该避免到处使用全局变量，而尽可能优先保证局部变量的内聚性。从这个角度来讲，Perl 式的设计是不能借鉴的（当然如果是一次性的脚本，这样倒是很方便）。Ruby 式的设计是比较合理的，但按这个设计写出来的程序可能到处是记号，丧失了程序的美感（这只是我主观的感受）。因此 crowbar 采用了 PHP 风格的 `global` 语句的设计。

3.1.5　语句与结构控制

crowbar 与 C 语言一样，有 `if`、`while`、`for` 等结构控制语句。
与 C、C++、Java 等语言有以下两处比较大的区别：

- crowbar 中不允许出现悬空 else（花括号 {} 是强制书写的）；
- 因为不允许悬空 else，所以引入了 `elsif` 语句。

具体来说是下面这样的形式：

```
# if 语句的例子
if (a == 10) {
    # a == 10 时执行
} elsif (a == 11) {
    # a == 11 时执行
} else {
    # a 不为 10 也不为 11 时执行
}
```

```
# while 语句的例子
```

```
while (i < 10) {
    # i 比 10 小时，此处循环执行
}
```

```
# for 语句的例子
for(i = 0; i < 10; i = i + 1) {
    # 这里循环 10 次
}
```

此外，在 crowbar 中也可使用下列语句，其意义与 C 语言相同。

- `break`：从最内层的循环中跳出。
- `continue`：跳过最内层循环中剩余的代码。
- `return`：从函数退出，并将后面的值作为返回值返回。

`break` 或 `continue`，最好能像 Java 那样附加一个标签，但当前版本还没有这个功能（book_ver.0.3 实现了标签功能）。

补充知识　elif、elsif、elseif 的选择

C 等语言中，`if` 语句允许没有花括号的写法（也称作悬空语句），也可以像下例这样用 `else if` 排列书写。

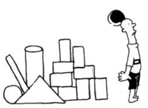
```
if (a < 10) {
    ⋮
} else if (a < 20) {
    ⋮
} else if (a < 30) {
    ⋮
} else {
    ⋮
}
```

C 或 Java 虽然设置了这种特别的结构控制语法，但偶尔也有初学者会误解其意义，以为 `else if` 不是一个专用语句，而是 `else` 语句后省略花括号又写的一个 `if` 语句。说起来在工作中的确会遇到很多项目，在编码规范中明确规定了“禁止省略花括号”，这样就可以放心地去写 `else if` 了。

crowbar 中直接废弃了悬空语句，无法书写上述形式的 `else if`，为此特别引入了 `elsif`。不过不同语言对于 `elsif` 的设计都不太一样，实在让人有些头疼。

B Shell、Python、C 预处理器	`elif`
Perl、Ruby、MODULA-2、Ada、Eiffel	`elsif`
Visual Basic、PHP	`elseif`

因为 crowbar 的目标就是成为“Perl 那样的东西”，所以我就私自决定采用 `elsif` 了。



3.1.6 语句与运算符

首先，crowbar 支持以下形式的常量作为语句。

- 整数字面常量，如 123 等。
- 实数字面常量，如 123.456 等。
- 字符串字面常量。双引号包裹的字符串，如 "abc" 等。

另外变量也可以作为语句。

进而可以和运算符结合构成更复杂的语句，当然还支持括号。

crowbar 可使用的运算符如表 3-1 所示（按运算优先级排序）。

表 3-1 crowbar 可使用的运算符

<table>
<tr><td>-（单目取负）</td><td>符号的反转</td></tr>
<tr><td>* / %</td><td>乘法、除法、求余</td></tr>
<tr><td>+ -</td><td>加法、减法</td></tr>
<tr><td>> >= < <=</td><td>大小比较</td></tr>
<tr><td>== !=</td><td>同值比较</td></tr>
<tr><td>&&</td><td>逻辑与</td></tr>
<tr><td>||</td><td>逻辑或</td></tr>
<tr><td>=</td><td>赋值</td></tr>
</table>

% 运算也可以用在实数上，本质上是在内部调用了 C 的函数 fmod()。

无论 C 语言还是 crowbar，都没有用常量直接表示负数。想使用负数时，可以使用单目取负符 -。

而与 C 语言一样，&&、|| 都是短路运算符。也就是说，像下面这样的条件语句：

```
if (a < 10 && b < 20) {
    ⋮
```

当 a < 10 的条件不成立时，不再判断 b < 20 这一条件语句（已经短路，所以表达式无论真伪都不会在 if 语句中执行）。

3.1.7 内置函数

内置函数是 crowbar 最开始就包含的用 C 语言编写的函数。crowbar 当前版本的内置函数如表 3-2 所示。

表 3-2
crowbar book_ver.0.1 的内置函数

函数名	功能
print(arg)	显示 arg。arg 的类型可以是整数、实数、字符串
fopen(filename, mode)	打开一个文件，返回文件指针。mode 的可选参数与 C 语言的 fopen() 一样（其实就是原封不动的传给了 C 语言）
fclose(fp)	传入 fp 即关闭文件
fgets(fp)	从 fp 中读出一行字符串并返回
fputs(str, fp)	向 fp 输出字符串，输出时不会自动添加换行

显而易见，基本上所有文件操作函数的设计都沿袭了 C 语言的 stdio.h。只是因为 crowbar 有字符串类型，所以 `fgets()` 等的用法会稍有不同。

此外，`fopen()` 返回的类型是 crowbar 才有的“原生指针型”。上例中只是单纯指向 C 的 `FILE*`，但是这个类型的特殊之处远不止于此。比如用内置函数实现 GUI 时，创建一个打开新窗口的函数 `create_window()`，其返回值应当能表示一个“窗口”，此时就可以考虑使用原生指针型来实现。

crowbar 中已经默认声明了 `STDIN`、`STDOUT`、`STDERR` 等全局变量，分别对应 C 语言中的 `stdin`、`stdout`、`stderr`。

3.1.8 让 crowbar 支持 C 语言调用

考虑到 crowbar 的用途之一是扩展应用程序，那么应当让 C 语言编写的其他应用程序可以很容易地调用 crowbar 解释器。

代码清单 3-3 是与当前版本 crowbar 所属的 main.c 基本一样的代码段。调用里面这些函数，需要用 `#include` 包含 CRB.h 文件。

代码清单 3-3
crowbar 被 C 语言调用

```
CRB_Interpreter     *interpreter;
FILE *fp;
/* 中间省略 */

/* 生成 crowbar 解释器 */
interpreter = CRB_create_interpreter();

/* 将 FILE* 作为参数传递并生成分析树 */
CRB_compile(interpreter, fp);

/* 运行 */
CRB_interpret(interpreter);
```

```
/* 运行完毕后回收解释器 */
CRB_dispose_interpreter(interpreter);
```

3.1.9 从 crowbar 中调用 C 语言（内置函数的编写）

反过来，从 crowbar 中调用 C 语言的函数（内置函数）也同样容易。

首先用 `#include` 包含面向开发人员的头文件 CRB_dev.h，像下面这样表示 C 函数：

```
CRB_Value hoge_hoge_func(CRB_Interpreter *interpreter,
                         int arg_count, CRB_Value *args)
{
    /* 中间省略 */
    return value;
}
```

这里调用的 `interpreter` 是指向解释器的指针，`arg_count` 代表向该函数传递的参数的数量，`args` 是参数的值（CRB_Value 类型详见 3.3.8 节）。

crowbar 是无类型语言，因此参数的数量与类型的检查都必须在内置函数进行。

通过这种方式制作出的 C 函数，通过 `CRB_add_native_function()` 函数即可注册到解释器中，成为 crowbar 的内置函数。

```
/* 将 C 的函数 hoge_hoge_func 注册为一个 crowbar 可以调用的内部函数
   并命名为 hoge_hoge */
CRB_add_native_function(interpreter,
                        "hoge_hoge", hoge_hoge_func);
```



3.2 预先准备

crowbar 的语言处理器有一定的行数规模（最终版有 8000 行左右），因此应当预先约定编码规范，并准备好底层的库。

那么让我们暂时离开语言处理器，先准备下面这些事项吧。

3.2.1 模块与命名规则

crowbar 由以下 3 个模块构成：

- crowbar 主程序（CRB）
- 内存管理模块（MEM）
- Debug 模块（DBG）

括号中的 CRB、MEM 等是模块名。

这里我所指的模块，即可以完成某些特定功能的程序块。一个模块中基本都会包含多个 .c 文件。

MEM 与 DBG 均为通用模块，并不是 crowbar 专用的。代码分别位于 crowbar 文件夹下的 memory、debug 子文件夹中。

C 语言中没有 C++ 和 C# 的命名空间，也没有 Java 中的包机制，因此必须制定命名规范来避免可能出现的命名冲突。因此我们使用以下的命名规范。

1. 模块必须有前缀3个字母的缩写（如：`CRB`）。
2. 类型名，以大写字母开始，并使用下划线连接单词（如：`CRB_Interpreter`）。
3. 变量名/函数名，全部使用小写字母，使用下划线连接单词（如：`alloc_expression()`）。
4. 宏命名为全大写字母，使用下划线连接单词（如：IDENTIFIER_TABLE_ALLOC_SIZE）。但如果是带参数的宏，特别是具有函数功能的部分，则要遵循函数的命名规则（如：`small(a, b)`）。
5. 模块中向外公开的函数，命名以模块名（大写字母）+ 下划线作为前缀（如：`CRB_create_interpreter()`）。
6. 模块中不对外公开的函数，如果函数的作用域跨文件时，则函数名以模块名（小写字母）+ 下划线作为前缀（如：`crb_alloc_expression()`）。
7. 函数外的静态变量名以`st_`作为前缀（如：`st_string_literal_buffer`）。

各模块中向外部公开的接口需要做成**公有头文件**的形式，在头文件中定义了公开函数以及调用模块所需的类型。比如 crowbar 中，想使用 crowbar 解释器就需要包含 CRB.h，而编写 crowbar 的内置函数则需要包含 CRB_dev.h。

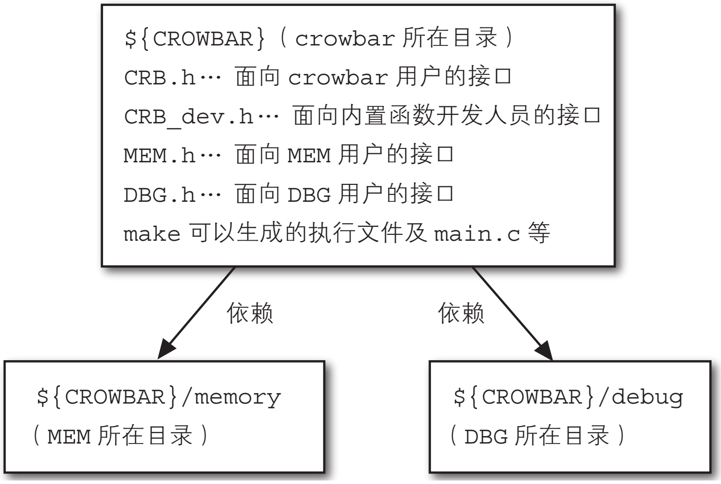
各模块内部使用的类型、宏、函数等，则可以声明为**私有头文件**。比如在 crowbar 中，crowbar.h 就是一个私有头文件，其中声明的类型名或宏无需附加 `CRB_` 前缀（因为外部是接触不到的）。但是函数与全局变量，为了以防万一还是需要加上 `crb_` 前缀的。

2.3.1 节后的补充知识中曾写道，所有的头文件应当尽量只用一个 #include（前提是已经加入了防止多重定义的处理）。因此大多数情况下，私有头文件内部可以用 #include 包含公有头文件，反之则不行。内部文件中使用公共信息，而外部文件中则不能含有私有信息，这应该不难理解。

经过上述的处理，各模块的内部细节都可以对其他模块实现隐藏（即面向对象中常提到的封装概念）。此外，在 C 语言中，头文件修改后包含该头文件的源代码都需要重新编译。将头文件划分为公有及私有，只要保证公有头文件不修改，那么用户利用公有头文件编写的程序也就无需重新编译了。

crowbar 的模块与目录结构如图 3-2 所示。

图 3-2 crowbar 的模块与目录结构



3.2.2 内存管理模块 MEM

经常使用 C 的程序员应该深有体会，用 C 语言编程时，难免会遇到诸如内存损坏（memory corruption）BUG、忘记释放内存引起内存泄漏、引用的内存区域被释放让 BUG 难以重现等问题，总之围绕内存经常会发生很多让人讨厌的 BUG。

而由于 crowbar 还设置有字符串型的变量，可以用 + 运算符连接字符串，因此我们必须配置某种垃圾回收机制。比如：

```
a = "a" + "b" + "c"
```

这个语句运行的时候，首先执行 "a" + "b" 语句生成字符串 "ab"，然后

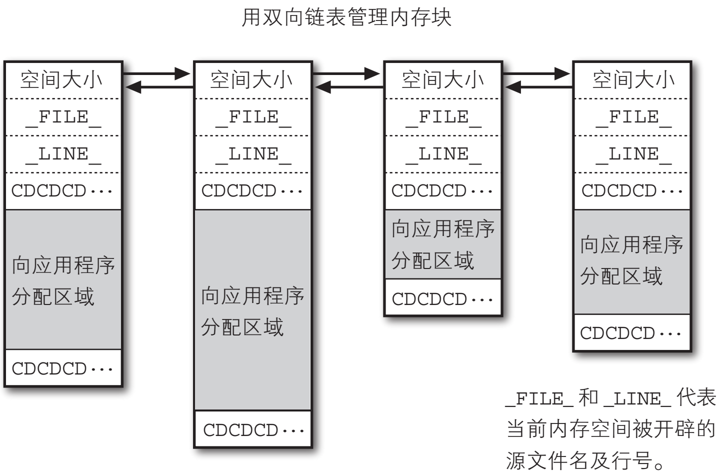
为了继续生成 "abc"，"ab" 的内存空间必须自动释放。具体的运行过程请参考 3.3.11 节。正是由于运行时需要运行很多这样繁复的处理，很容易出现 BUG，所以需要有一种方法来确认内存中到底发生了什么。

基于上述理由，我制作了一个具备下列功能的内存管理模块。模块名为 MEM，按之前的命名规范，所有的公共函数都以 MEM_ 为前缀。

1. 通过MEM_malloc()可以分配内存空间，内存空间开始处默认填充有0xCC。常规的malloc()函数开辟的内存空间值为0的情况很多，因此很容易遗漏初始化过程。而0xCC毫无疑问是个无意义的值，这样就可以确保能够检查出被遗漏的初始化过程。
2. MEM_realloc()用于扩充内存空间时，也会默认填充0xCC。
3. 开辟的内存空间用MEM_free()释放时，被填充的0xCC也会被释放。由此可以较早地发现由于引用被释放的内存空间而引起的BUG。
4. MEM模块会以链表形式保存所有开辟的内存空间，可以使用MEM_dump_block()将其转储。转储后可以将MEM_malloc调用位置的源文件名及行号显示出来。
   用malloc()开辟的内存空间，在不用的时候一定要用free()释放，这是我们在编程时一定要遵守的一个准则。那么如果在程序结束时调用MEM_dump_block()仍然看到有结果输出的话，就可以断定某处发生了内存泄漏。
5. MEM_malloc()开辟的内存空间在传递给程序使用时，空间前后会加上0xCD的记号，检查这些记号就可以知道由于数组越界等问题引起的内存损坏程度了。
   这个检查还需要配合使用MEM_check_block()、MEM_check_all_blocks()等函数。

内存管理模块 MEM 会以图 3-3 的形式管理内存。

图 3-3
通过 MEM 管理内存



很简单的模块，功能虽然简单，但对于 BUG 的检查非常有用。

对于动态开辟的内存空间，经常会先开辟若干个小型的区域，然后将这些区域一起释放。分析树的节点就是典型的例子。开辟空间会一点一点地进行，释放则是一次性的。对此，MEM 模块引入了存储器（storage），作为开辟内存的常规工具。

1. 由MEM_open_storage()生成一个新的存储器。
2. MEM_storage_malloc()可以接受存储器和空间的大小作为入口参数，并返回所请求大小的内存空间。
3. 由MEM_dispose_storage()将存储器内所有的内存空间全部释放。

MEM_storage_malloc() 会将 MEM_open_storage() 开辟的较大内存空间，从起始处按照请求的尺寸一次性全部返回。因此无法对其中的子空间单独释放，也不能通过 realloc() 扩展空间。

补充知识 valgrind

我们手动实现了模块 MEM，它可以检查由于 C 语言操作内存而引起的 BUG，其实也有很多其他工具具备同样的功能。

以前这类工具大都是需要付费的，不过在 Linux 环境下，可以使用自由软件 valgrind（许可证为 GPL）。

通常我们使用下面的方式启动程序（% 是命令行提示符）。

```
% crowbar test.crb
```

而执行如下指令的话，

```
% valgrind crowbar test.crb
```

可以帮助我们检查是否忘记释放内存（或内存泄漏），以及是否在程序开辟的内存空间外部进行了写入。

可能有读者会问，有这么方便的工具为什么还要制作 MEM 模块呢？实际上，我在写 MEM 模块时完全不知道有 valgrind 这个工具，算是重复发明轮子了。不过自己实现一个这样的工具也是有好处的吧。

valgrind 的详细内容，请参考官方主页 http://valgrind.org/。

补充知识 富翁式编程

MEM 模块中，在应用程序所使用的内存空间前后分别加上了管理专用空间。比如开发环境中 int 型或者指针一般占用 4 字节，double 型一般占用 8 字节，而其管理空间前面占用 24 字节，后面则有 8 字节（包含校验信息）。

那么如果生成很多对象时，肯定会浪费很多内存空间。

然而对于现在的电脑来说，这种程度的浪费简直是微不足道的。与其冥思苦想节省内存空间或提高处理速度的小技巧，倒不如专注于如何提高开发效率。这种编程方式就叫作**富翁式编程***。

* 参考 URL：http://www.pitecan.com/fugo.html

crowbar 的实现就非常之“富翁”。比如用 crowbar 书写的程序中会出现 `hoge_piyo_foo_bar` 这样一个变量。对于现代编程语言来说，这样长的变量名或函数名是很常见的。在程序中，`hoge_piyo_foo_bar` 变量名可能还会出现若干次，crowbar 的解释器将会预先对所有出现的变量名分配好空间，当然这都是要消耗内存的，最终只需要用 `strcmp()` 简单地对当前变量名做一致性检查就可以了。另外，在检索变量或函数时，都采用线性检索。

这样设计当然可能会出现运行速度慢的情况，即便如此，等到状况发生时再想办法优化也不迟。在前期优先考虑的应该是如何让程序更加容易编写、理解起来更加简单。

补充知识　符号表与扣留操作

刚刚提到过，无论是内存空间还是处理速度，crowbar 的内部实现都是比较“富翁式”的。那么如果在此基础上想要进一步提高运行效率的话要怎样做呢?

正如上文所述，crowbar 对于程序中多次出现的变量名等，会分别开辟空间将其保存。如果变量名较长时比较浪费，因此将同名变量整合为一处保存，不失为一个提高效率的方法。

具体来说，程序中会存在一个函数，为所有出现的特征符建立数据结构，新出现的特征符如果已经被记录则会返回其指针，如果尚未记录则会新录入并返回指针。这样的操作称为**扣留**（intern）。对一个标识符进行扣留操作时，无需判别该标识符是局部变量还是全局变量，或是函数名（当然进行判别也无妨）。

对程序中出现的所有的标识符一一进行扣留操作的话，在判断两个标识符是否为同一个时，只需要比较它们的指针就可以了。这比用 `strcmp()` 更快。

而 crowbar 对于局部变量、全局变量和函数则分别使用链表进行管理。一旦语句中出现变量名时，将从链表头部开始检索（采用线性检索）。如果要优化这个部分，可以考虑引入缓存、树或二分法查找等。刚才提到的这些数据结构及算法，都是编程语言语言处理器普遍使用的，读者可以自行查阅相关图书或网站。

一般来说，我们将编译器保存变量名、函数名的数据结构称为“符号表”。

3.2.3　调试模块 DBG

DBG 是调试时使用的模块，具备若干功能，在 crowbar 的代码中使用的话，只需要调用宏 `DBG_assert()` 及 `DBG_panic()` 即可。

```
/* 断言这里 a 的值应该为 5 */
DBG_assert(a == 5, ("a..%d", a));
```

这样书写的话，当 a == 5 这一条件不成立时，程序会将该处的源代码行号输出并执行 abort()。第二个入口参数则是将想要输出的东西传递给 printf() 并格式化（因为宏无法使用可变长度的参数，因此需要从第二参数起全部用括号括起来）。

DBG 的输出目标可以通过 DBG_set_debug_write_fp() 函数进行更改，标准输出目标是 stderr。而输出目标无论如何更改，stderr 仍然会保留一份同样的信息。因此如果不做任何更改的话，会看到 stderr 输出的是两行同样的信息。

DBG_panic() 函数可以书写在一些程序不应该进入的分支处。典型的例子就是 switch case 的 default 分支，如：

```
/* 变量 operator 通过 switch case 判断分支条件
  default 分支在正常情况下不应当进入 */
default:
    DBG_panic(("bad case...%d", operator));
```

与 DBG_assert() 一样，用两层括号包裹，最终会通过 printf() 格式化输出。

DBG_assert() 与 DBG_panic() 都是宏，只要在定义 #define DBG_NO_DEBUG 的状态下编译，就可以完全删除执行文件中的调试部分。



3.3 crowbar ver.0.1 的实现

预先准备已经差不多了，终于可以开始阅读 crowbar book_ver.0.1 的代码了。

3.3.1 crowbar 的解释器——CRB_Interpreter

一般来说，程序的数据结构要比运行流程更加重要，因此我们就从 crowbar 解释器所用的结构体 CRB_Interpreter 开始看起。

想使用 crowbar，首先需要生成解释器，然后将解释器的源码传递给编译器（生成分析树），就可以运行了。



解释器的定义如下所示（位于 crowbar.h）。注意 CRB.h 中公开的 CRB_Interpreter 是这个结构体的不完全定义，下面这个结构体定义本身对外是隐藏的。

```
struct CRB_Interpreter_tag {
    MEM_Storage         interpreter_storage;
    MEM_Storage         execute_storage;
    Variable            *variable;
    FunctionDefinition  *function_list;
    StatementList       *statement_list;
    int                 current_line_number;
};
```

解释器会保存以下内容：

1. 与解释器相同生命周期的**MEM_Storage**（**interpreter_storage**）

不再需要分析树时，需要将其释放，如 3.2.2 节所述，可以使用内存管理模块 MEM 提供的存储器功能来管理。

interpreter_storage 存储器，在解释器生成时被生成，解释器废弃的同时被释放。通过 CRB_Interpreter 自己来开辟这个存储器。

该存储器在内存中的开辟，是通过位于 util.c 中的 crb_malloc() 工具函数实现的。

2. 运行时使用的**MEM_Storage**（**execute_storage**）

execute_storage 是运行时使用的存储器。不过由于运行时必备的数据结构大多数都没有固定的释放顺序，因此 execute_storage 现阶段主要用于存放全局变量。

3. 全局变量链表（**variable**）

Variable 结构体的定义如下所示：

```
typedef struct Variable_tag {
    char        *name;  /* 变量名 */
    CRB_Value   value;  /* 变量值 */
    struct Variable_tag *next;  /* 指向下一个变量的指针 */
} Variable;
```

首先这个结构体中有 next 这一成员，这是为了构建链表用的。CRB_Interpreter 的成员 variable 保存在最开头。通过这样的链表，可以得到所有的全局变量。

crowbar 中变量是在首次赋值时生成的，因此在运行时会有变量逐一进入，链表也会越来越长。

`Variable` 结构体的 `name` 成员顾名思义会保存变量名，而 `value` 成员则会保存该变量的值，具体请参考 3.3.8 节对 `CRB_Value` 的说明。

4. 函数定义链表（`function_list`）

`function_list` 是记录 crowbar 中编写函数的链表。语法解析时会创建这个 `function_list` 以及下面的 `statement_list`。

`FunctionDefinition` 类型的定义如下所示：

```
typedef enum {
    CROWBAR_FUNCTION_DEFINITION = 1,  /* crowbar中定义过的函数 */
    NATIVE_FUNCTION_DEFINITION        /* 内置函数 */
} FunctionDefinitionType;

typedef struct FunctionDefinition_tag {
    char                *name;   /* 函数名 */
    FunctionDefinitionType       type;  /* 函数的类型 */
    union {
        struct {
            ParameterList         *parameter;  /* 参数的定义 */
            Block                 *block;      /* 函数的主体 */
        } crowbar_f;
        struct {
            CRB_NativeFunctionProc     *proc; /* 后文详述 */
        } native_f;
    } u;
    struct FunctionDefinition_tag      *next;  /* 链表用 */
} FunctionDefinition;
```

`FunctionDefinition` 类型的 `type` 成员中，会区分 crowbar 定义的函数以及内置函数。crowbar 定义的函数会使用下面联合体 u 的 `crowbar_f`，而内置函数则会使用 `native_f`。通过这种方法，可以让没有继承概念的 C 语言实现类似继承的功能（具体方法之后会慢慢提到）。

crowbar 定义的函数会通过 `crowbar_f` 成员保存其函数参数及函数主体（执行语句）。`ParameterList` 结构体如下所示，会将变量名做成链表并保存（crowbar 是无类型语言，无需保存变量类型）。

```
typedef struct ParameterList_tag {
    char        *name;                /* 变量名 */
    struct ParameterList_tag *next;   /* 链表所用指针 */
```

```
} ParameterList;
```

用 block 成员保存函数的执行语句。block 是 Block 类型的结构体，Block 类型的定义如下所示：

```
typedef struct {
    StatementList       *statement_list;
} Block;
```

StatementList 正如其名称，是语句的链表。其结构体内容请参考第 5 项。

5. 语句链表（statement_list）

statement_list 是语句的链表。其类型与上面的 Block 结构体保存时所用的 StatementList 类型相同。无论是函数定义 { } 内的语句，还是顶层结构中的语句，从内部来讲都保存在 StatementList 中。

crowbar 的解释器在语法解析后，顶层结构的语句，也就是 statement_list 最开头保存的语句会按照顺序开始执行。

6. 编译时当前的行号（current_line_number）

出现错误信息需要行号。current_line_number 可以在编译时显示当前的行号。

current_line_number 在编译结束后不会再使用。运行时如果发生错误当然也需要显示行号，这里的行号保存在分析树的语句节点中。

补充知识　**不完全类型**

CRB_Interpreter 类型结构体是在 crowbar 的私有头文件 crowbar.h 中定义的。不过这是供解释器内部使用的数据结构，不应该向外部公开。

而生成解释器的函数 CRB_create_interpreter()，它的原型定义则是在公有头文件 CRB.h 中：

```
CRB_Interpreter *CRB_create_interpreter(void);
```

CRB_create_interpreter() 返回值的类型为 CRB_Interpreter*，为了支持这样的原型定义必须首先定义 CRB_Interpreter 结构体。但是我们不能把解释器的内部定义直接拿出来放在公有头文件中。

应对这种情况可以使用**不完全类型**。公有头文件中只定义结构体的标识符，实际的定义是由私有头文件传递给公有头文件的。

比如上面的 CRB_Interpreter 类型，在 CRB.h 中可以做如下的标识符定义，并用 typedef 命名。

```
typedef struct CRB_Interpreter_tag CRB_Interpreter;
```

这种状态的 `CRB_Interpreter` 就是不完全类型。

不完全类型只能使用指针，即指向不完全类型的指针的变量无法被声明，不完全类型本身也无法声明变量，对不完全类型无法使用 `sizeof`。因此，我们是无法知道一个不完全类型的大小的，当然也无法引用其成员。

而 crowbar.h 是类型初始定义所在的地方，因此没有这些限制（详见 3.3.1 节）。这里的 `CRB_Interpreter` 类型就不是不完全类型。

上述都是 C 语言编程中必须掌握的一些技巧，但我意外地发现似乎了解的人不多，因此写了下来。

3.3.2 词法分析——crowbar.l

crowbar 的 lex 定义文件是 crowbar.l。源代码的摘录版本如代码清单 3-4 所示。

代码清单 3-4
crowbar.l

```
%{
 省略 C 编码部分
%}
 开始条件
%start COMMENT STRING_LITERAL_STATE
%%
 保留字的定义
<INITIAL>"function"     return FUNCTION;
<INITIAL>"if"           return IF;
 省略其他的保留字定义
 符号类的定义。LP，RP 是 Left/Right Paren 的缩写
 LC，RC 是 Left/Right Curly（花括号）的缩写
<INITIAL>"("            return LP;
<INITIAL>")"            return RP;
<INITIAL>"{"            return LC;
<INITIAL>"}"            return RC;
 省略其他的符号定义
 标识符（变量名、函数名等）
<INITIAL>[A-Za-z_][A-Za-z_0-9]* {
    yylval.identifier = crb_create_identifier(yytext);
    return IDENTIFIER;
}
 数值。整数类型与实数类型分别处理，与 mycalc.y 相同
<INITIAL>([1-9][0-9]*)|"0" {
    Expression  *expression = crb_alloc_expression(INT_EXPRESSION);
    sscanf(yytext, "%d", &expression->u.int_value);
    yylval.expression = expression;
    return INT_LITERAL;
```

```
}
<INITIAL>[0-9]+\.[0-9]+ {
    Expression  *expression = crb_alloc_expression(DOUBLE_EXPRESSION);
    sscanf(yytext, "%lf", &expression->u.double_value);
    yylval.expression = expression;
    return DOUBLE_LITERAL;
}
```

定义字符串开始

```
<INITIAL>\" {
    crb_open_string_literal();
    BEGIN STRING_LITERAL_STATE;
}
<INITIAL>[ \t] ;
```

遇到换行符则增加行号

```
<INITIAL>\n {increment_line_number();}
```

定义注释的开始

```
<INITIAL>#     BEGIN COMMENT;
```

如果不符合上述定义，则为非法字符并报错

```
<INITIAL>.      {
    char buf[LINE_BUF_SIZE];

    if (isprint(yytext[0])) {
        buf[0] = yytext[0];
        buf[1] = '\0';
    } else {
        sprintf(buf, "0x%02x", (unsigned char)yytext[0]);
    }

    crb_compile_error(CHARACTER_INVALID_ERR,
                      STRING_MESSAGE_ARGUMENT, "bad_char", buf,
                      MESSAGE_ARGUMENT_END);
}
<COMMENT>\n     {
    increment_line_number();
    BEGIN INITIAL;
}
<COMMENT>.      ;
<STRING_LITERAL_STATE>\"        {
    Expression *expression = crb_alloc_expression(STRING_EXPRESSION);
    expression->u.string_value = crb_close_string_literal();
   yylval.expression = expression;
   BEGIN INITIAL;
   return STRING_LITERAL;
}
<STRING_LITERAL_STATE>\n        {
   crb_add_string_literal('\n');
   increment_line_number();
```

```
}
<STRING_LITERAL_STATE>\\\"    crb_add_string_literal('"');
<STRING_LITERAL_STATE>\\n     crb_add_string_literal('\n');
<STRING_LITERAL_STATE>\\t     crb_add_string_literal('\t');
<STRING_LITERAL_STATE>\\\\    crb_add_string_literal('\\');
<STRING_LITERAL_STATE>.       crb_add_string_literal(yytext[0]);
%%
```

crowbar.l 与之前计算器的例子 mycalc.l 相比（代码清单 2-1）要长很多，但本质上没有太大变化，只是使用了新的**开始条件**功能。与 mycalc.l 不同的是，在 crowbar.l 的大部分规则前都要书写 `<INITIAL>`，这就是开始条件。

在 crowbar 中，开始条件主要用于分割注释与字面常量（literal）。

crowbar 的注释由 # 开头直到行尾，可以用简单的正则表达式 `#.*$` 将其分割，分割出的注释暂时保存在全局变量 `yytext` 中。

在以前的 lex 处理器中，给 `yytext` 分配了一个固定大小的 `char` 数组，而且数组的大小是无法扩展的。不过包含 flex 在内，最近发布的 lex 处理器中，`yytext` 已经更改为 `char*`，当一个很长的记号进入时，也可以动态扩展储存空间，注释最终也是要被丢弃的，如果还在这里特意去扩展存储空间的话，就显得有点笨了。

crowbar 的字面常量与 C 语言一样，是包含 `\n` 和 `\t` 的，还可以通过 `\"` 显示双引号本身，因此简单通过 `\".*\"` 的正则表达式规则进行匹配是不行的。

应对这种情况可以使用开始条件。在动作中书写 `BEGIN COMMENT` 切换 lex 的状态，其对应的规则就变成后面用 `<COMMENT>` 开始的部分了。lex 使用 `INITIAL` 定义了开始条件的初始状态。

crowbar.l 中，通过下面的处理将注释读入并丢弃。

```
77: <INITIAL>#     BEGIN COMMENT;
     中间省略
92: <COMMENT>\n     {
93:     increment_line_number();
94:     BEGIN INITIAL;
95: }
96: <COMMENT>.      ;
```

代码第 77 行，`INITIAL` 状态下如果有 # 进入，则转换为 `COMMENT` 状态。crowbar 的注释由 # 开始直至行尾，因此在 `COMMENT` 状态下如果遇到换行则切换回 `INITIAL` 状态（第 92 ～ 95 行的 `increment_line_number()` 会在后文详述）。所以，`COMMENT` 状态下会将除换行符以外的字符全部丢弃（第 96 行）。

关于字符串的字面常量处理，开始时调用 crb_open_string_literal()，中间的字符通过 crb_add_string_literal() 追加，最后通过 crb_close_string_literal() 结束一个字符串的处理。

在这个过程中，字符串会被保存在 string.c 的 st_string_literal_buffer 这一 static 变量中。我本人对于使用 static 变量还是有些抵制的，但是鉴于我们只能把 yacc/lex 作为工具来使用，即使有什么想法也无法修改（至少老版本是不允许修改的），因此 crowbar 最终还是允许在编译器中使用静态变量的（请参考本节的补充知识）。

语言处理器在编译时如果发生错误，需要显示错误信息，因此错误信息中必须包含行号。

在 crowbar.l 中的对应处理是，每换行一次，行号都会进行计数。具体来说有以下三处：

```
<INITIAL>\n {increment_line_number();}
  ⋮
<COMMENT>\n     {
    increment_line_number();
    BEGIN INITIAL;
}
  ⋮
<STRING_LITERAL_STATE>\n        {
    crb_add_string_literal('\n');
    increment_line_number();
}
```

这里进行计数的行号保存在 CRB_Interpreter 的 current_line_number 中。

补充知识　静态变量的许可范围

如上文所写，在词法分析中，正在读入的字符串会保存在 string.c 的 st_string_literal_buffer 中。在编译时，当前的编译器会保存在 util.c 的 st_current_interpreter 中。而在 yacc/lex 中，还会使用 yytext 等全局变量。

使用如此多的静态变量，首当其冲会遇到的就是多线程问题。

当下多线程的程序已经很普及了，静态变量可以在多线程之间共享，因此多个线程如果同时进行编译的话可能会引发问题。

不过一般来说，编译过程都是一下子就能结束的，因此在这样短的时间内通过加一个全局锁的方式就可以解决问题了。正因为有这样既简单又实用的方法，crowbar 才放心地允许在编译过程中使用静态变量。而静态变量仅在编译过程中被使用，一旦程序开

始运行后就无法使用了。具体来说，由于 MEM 模块会静态地保存内存块链表，这原本就是以调试为目的创建的链表，可以随时删除，而由于 MEM 模块位于系统最底层，因此可以对 `MEM_malloc()` 的运行进行加锁处理。

使用静态变量还会造成另一个问题，就是编译器无法递归运行。比如 crowbar 中的库函数都书写在独立文件中，在 C 语言中理论上只需要用 `#include` 就能很简单地把功能包含进来。但是实际应用中就会发现，使用 `#include` 包括进来的独立文件，在编译过程中如果再开始一个解析器，之前的静态变量就会被覆盖。

标准版的 yacc/lex，由于使用了 yytext 等全局变量，因此也无法支持多线程或使用递归。不过 bison 有单独的扩展可以让其支持多线程。*

* 在后面写到的 Diksam 中，具备与 C 语言的 `#include` 相对应的功能 `require`，同样由于解析器不能进行递归，会在解析完一个文件后，才开始解析被 `require` 的文件。

3.3.3 分析树的构建——crowbar.y 与 create.c

crowbar 中 yacc 的定义文件为 crowbar.y。从构成来说，与计算器版的 mycalc.y 没有什么变化。

但是在计算器中，归约是在实际进行计算时才进行的，而 crowbar.y 则是在构建分析树时进行的。

比如一个加法算式（如 `10 + a`）会按照以下的规则构建：

```
additive_expression
    （中间省略）
    | additive_expression ADD multiplicative_expression
    {
        $$ = crb_create_binary_expression(ADD_EXPRESSION, $1, $3);
    }
```

这里的 `additive_expression` 对应 mycalc.y 中的 `expression`，`multiplicative_expression` 对应 `term`。

在动作中，`crb_create_binary_expression()` 被调用，其实际运行代码在 create.c 中。这个函数负责常量折叠（参考 3.3.4 节），所以稍微有些复杂，将这部分以外的核心代码精简一下，可以看到这个函数的主要逻辑如下所示：

```
Expression *
crb_create_binary_expression(ExpressionType operator,
                             Expression *left, Expression *right)
{
    Expression *exp;
    exp = crb_alloc_expression(operator);
    exp->u.binary_expression.left = left;
    exp->u.binary_expression.right = right;
```

```
    return exp;
}
```

crb_alloc_expression() 将开辟一个存放 Expression 类型结构体的内存空间，并将其返回。Expression 类型在 crowbar.h 中的定义如下所示：

```
struct Expression_tag {
    ExpressionType type;        ←表示表达式的类别
    int line_number;
    union {                     ←用联合体保存不同种类对应的值
        CRB_Boolean             boolean_value;
        int                     int_value;
        double                  double_value;
        char                    *string_value;
        char                    *identifier;
        AssignExpression        assign_expression;
        BinaryExpression        binary_expression;
        Expression              *minus_expression;
        FunctionCallExpression  function_call_expression;
    } u;
};
```

这个结构体在分析树中用来表示“表达式”的类型。与 CRB_Value 一样，用枚举类型的 ExpressionType 表示表达式的类型，用联合体保存各种类型对应的值。

ExpressionType 具体定义如下：

```
typedef enum {
    BOOLEAN_EXPRESSION = 1,     /* 布尔型常量 */
    INT_EXPRESSION,             /* 整数型常量 */
    DOUBLE_EXPRESSION,          /* 实数型常量 */
    STRING_EXPRESSION,          /* 字符串型常量 */
    IDENTIFIER_EXPRESSION,      /* 变量 */
    ASSIGN_EXPRESSION,          /* 赋值表达式 */
    ADD_EXPRESSION,             /* 加法表达式 */
    SUB_EXPRESSION,             /* 减法表达式 */
    MUL_EXPRESSION,             /* 乘法表达式 */
    DIV_EXPRESSION,             /* 除法表达式 */
    MOD_EXPRESSION,             /* 求余表达式 */
    EQ_EXPRESSION,              /* == */
    NE_EXPRESSION,              /* != */
    GT_EXPRESSION,              /* > */
    GE_EXPRESSION,              /* >= */
    LT_EXPRESSION,              /* <  */
    LE_EXPRESSION,              /* <= */
    LOGICAL_AND_EXPRESSION,     /* && */
```

```
    LOGICAL_OR_EXPRESSION,       /* || */
    MINUS_EXPRESSION,            /* 单目取负 */
    FUNCTION_CALL_EXPRESSION,    /* 函数调用表达式 */
    NULL_EXPRESSION,             /* null 表达式 */
    EXPRESSION_TYPE_COUNT_PLUS_1
} ExpressionType;
```

这其中从 ADD_EXPRESSION 到 LOGICAL_OR_EXPRESSION，都在使用联合体的 binary_expression 成员。

binary_expression 的类型是 BinaryExpression，其定义如下所示：

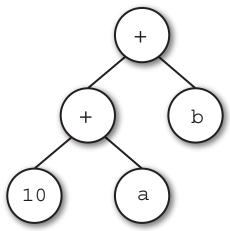
```
typedef struct {
    Expression  *left;
    Expression  *right;
} BinaryExpression;
```

crb_create_binary_expression() 最终返回这样构建出的 Expression 的指针，并在 crowbar.y 中将其赋值给 $$。

```
$$ = crb_create_binary_expression(ADD_EXPRESSION, $1, $3);
```

比如 10 + a + b 这样的语句，按上面的处理就会构建出如图 3-4 的分析树。

图 3-4
10+a+b 的分析树



那么，接下来是 crb_alloc_expression() 函数，这个函数只是简单地用 crb_malloc() 开辟 Expression 的空间，并且将 type 连同刚被设置的行号一起返回。

```
crb_alloc_expression(ExpressionType type)
{
    Expression  *exp;

    exp = crb_malloc(sizeof(Expression));
    exp->type = type;
    exp->line_number = crb_get_current_interpreter()->current_line_number;

    return exp;
}
```

在分析树中，不单是表达式，语句（statement）也很重要。构建语句的思路与表达式基本是一样的，关联的结构体定义从 crowbar.h 中摘录如下：

```
struct Statement_tag {
    StatementType       type;
    int                 line_number;
    union {
        Expression      *expression_s;   /* 表达式语句 */
        GlobalStatement global_s;        /* global 语句 */
        IfStatement     if_s;            /* if 语句 */
        WhileStatement  while_s;         /* while 语句 */
        ForStatement    for_s;           /* for 语句 */
        ReturnStatement return_s;        /* return 语句 */
    } u;
};
```

随便举一个联合体的例子，比如 `WhileStatement` 的定义如下所示：

```
typedef struct {
    Expression  *condition;  /* 条件表达式 */
    Block       *block;      /* 可执行块 */
} WhileStatement;
```

`Block` 在 3.3.1 节中已经出现过一次，即一段用花括号包裹的代码段类型。

`StatementType` 的一览表如下所示：

```
typedef enum {
    EXPRESSION_STATEMENT = 1,
    GLOBAL_STATEMENT,
    IF_STATEMENT,
    WHILE_STATEMENT,
    FOR_STATEMENT,
    RETURN_STATEMENT,
    BREAK_STATEMENT,
    CONTINUE_STATEMENT,
    STATEMENT_TYPE_COUNT_PLUS_1
} StatementType;
```

`BREAK_STATEMENT` 和 `CONTINUE_STATEMENT` 现阶段都没有信息需要保存（如果想像 Java 那样支持标签或 `break` 语句的话就需要保存了），因此没有对应的联合体成员。

3.3.4 常量折叠

比如

```
a = a + 10 * 2;
```

这样一个表达式，其实与

```
a = a + 20;
```

是同值的。诸如这种纯常量构成的表达式或部分表达式，在编译时提前被计算出来的处理方式叫作**常量折叠**(constant folding)。由于编译时这部分计算就已经完成了，所以能有效地提高运行速度。

crowbar 中也部分引入了常量折叠的处理。具体来说，在进行整数型与实数型相关的四则运算或者单目取负时会进行常量折叠 *。这部分代码本书不会进行详细说明，请参考 create.c 的 `crb_create_binary_expression()` 和 `crb_create_minus_expression()`。

* 字符串的加法不会做折叠处理。确实代码中有时候需要有长文本信息，但是按当前版本的处理方式，连接前的字符串是无法被释放的，也就是说其实这里是我偷懒了。

crowbar 中的常量折叠其实可以归入所谓程序最优化（ optimization ）的范畴，在语言的制作阶段考虑最优化本来为时尚早，但是由于 crowbar 的目标是类似 C 的语言，这种优化在部分场景中是必须的。比如 C 语言在一些特定地方只能写常量表达式（ `static` 变量的初始化或数组的大小指定等 * ），这些地方需要支持常量折叠的处理。

* ISO-C99 规范中，`auto` 的数组也可以不是常量。

程序优化是指通过不断调整程序以提高代码的运行速度，但是调整的结果到底是不是“最优”其实不好评判，因此有些人提出“最优化”这个用词是不恰当的。

3.3.5 错误信息

错误信息考虑到需要支持多语言 *，所以一般尽可能避免硬编码出现在代码中，最好以提供外部文件的方式实现。但是在开始设计 crowbar 这样的脚本语言时，我非常希望 crowbar 只通过一个可执行文件就能运行。比如我想去别的地方做一些文字处理工作，那么只需要把 crowbar 的唯一一个可执行文件拷入 U 盘就可以拿去用了。*

* 可能很多读者会认为显示错误信息只有英语就足够了，不过对于初级用户来说，英文的错误信息可能有点难以理解吧。更重要的理由是：我觉得自己的英语水平还不够。

* 不过一般公司对于数据的管理都是比较严格的，可能没有办法用 U 盘携带数据吧。

因此 crowbar 还是选择将错误信息硬编码在 error_message.c 的源文件中，参看代码清单 3-5。

代码清单 3-5
error_message.c

```
MessageFormat crb_compile_error_message_format[] = {
    { "dummy"},
    { "($(token) 附近有语法错误 )"},
    { " 错误的字符 ($(bad_char))"},
    { " 函数名重复 ($(name))"},
    { "dummy"},
};

MessageFormat crb_runtime_error_message_format[] = {
    { "dummy"},
    { " 找不到变量 ($(name))。"},
    { " 找不到函数 ($(name))。"},
    { " 传递的参数多于函数所要求的参数。"},
    { " 传递的参数少于函数所要求的参数。"},
    { " 条件语句类型必须为 boolean 型 "},
     ⋮
    { "dummy"},
};
```

通过这种方式，将来如果要支持多语言的话，可以简单地通过修改外部文件来实现。而设置 crb_compile_error_message 与 crb_runtime_error_message 这两个数组，是为了将编译时的错误信息与运行时的错误信息区分开来。这些数组的索引，与 crowbar.h 中对应的枚举型的值是一致的。

代码清单 3-6
crowbar.h 的错误信息枚举型定义

```
typedef enum {
    PARSE_ERR = 1,
    CHARACTER_INVALID_ERR,
    FUNCTION_MULTIPLE_DEFINE_ERR,
    COMPILE_ERROR_COUNT_PLUS_1
} CompileError;

typedef enum {
    VARIABLE_NOT_FOUND_ERR = 1,
    FUNCTION_NOT_FOUND_ERR,
    ARGUMENT_TOO_MANY_ERR,
    ⋮
    RUNTIME_ERROR_COUNT_PLUS_1
} RuntimeError;
```

代码清单 3-5 中的错误信息，包含了一个 $(token) 这样的字符串，这是错误信息中的可变部分。比如找不到一个名为 hoge 的变量时，如果能报出“找不

到变量 (hoge)”，要比只报“找不到变量”对用户更加友好。

为了实现错误信息中可以包含变量，显示错误信息时会调用下面的函数 crb_runtime_error()*。

* 只适用于运行出错。如果是编译出错，则调用 crb_compile_error()。

```
crb_runtime_error(expr->line_number,  /* 行号 */
                  VARIABLE_NOT_FOUND_ERR,  /* 枚举错误信息类别 */
                  STRING_MESSAGE_ARGUMENT, /* 可变部分的类型 */
                  "name", /* 可变部分的标识符 */
                  expr->u.identifier, /* 所要显示的值 */
                  MESSAGE_ARGUMENT_END);
```

第一个参数是行号（在 create.c 中 Expression 结构体中设定），下一个参数是错误类别（crowbar.h 中的 RuntimeError 枚举型），之后的三个参数为一组，对应错误信息的可变部分。STRING_MESSAGE_ARGUMENT 代表信息类型（相当于 printf() 中使用的 %s），name 即错误信息中的 $(name)，expr->u.identifier 则表示可变部分要显示的字符串。由于可变部分可以有多个，因此 crb_runtime_error 为一个变长参数，所以可以用 MESSAGE_ARGUMENT_END 表示参数输入结束。

当前版本无论是编译错误还是运行错误，显示错误信息后都会立即调用 exit() 终止程序。这样的处理其实还远远不够，如果用于扩展应用程序的话这样做更是致命的，因此应当参考 9.2.1 节加入异常处理机制。

补充知识　关于 crowbar 中使用的枚举型定义

比如在代码清单 3-6 中的 CompileError 类型，我特意将第一个元素 PARSE_ERR 设置为 1，而最后一个元素引入了名为 COMPILE_ERROR_COUNT_PLUS_1 的可变元素。

```
typedef enum {
    PARSE_ERR = 1,                    ←特意设置为1

    CHARACTER_INVALID_ERR,
    FUNCTION_MULTIPLE_DEFINE_ERR,
    COMPILE_ERROR_COUNT_PLUS_1        ←可变元素
} CompileError;
```

类似这样的处理方式不只有 CompileError 类型。比如用于显示 Expression 类别的 ExpressionType 类型也采用同样的构造。

特意这样设置的理由有下面几个。

1. 假如忘记进行初始化时，变量中被置入 0 的概率是非常高的，那么枚举类型如果从 1 开始的话，可以更早地发现异常状态。

2. 有了 COMPILE_ERROR_COUNT_PLUS_1 这个可变元素，就可以借助其遍历所有枚举元素，并在后续程序中利用这一特性进行更丰富的处理。

当然实际使用时，我发现这两处设置似乎也不是特别有效。

另外在错误信息中，错误信息数组的第一个和最后一个元素都是 dummy，这是为了防止在以后修改时只改了 crowbar.h 中的枚举类型，而忘记修改 error_message.c 中的对应错误信息，这样设置的话能在一定程度上自动检测这种遗漏（请参考 error.c 中的 self_check() 函数）。

```
MessageFormat crb_compile_error_message_format[] = {
    {"dummy"},
     :
    {"dummy"},
};
```

3.3.6 运行——execute.c

crowbar 程序的运行是从 CRB_interpret() 开始的，其函数实现如下：

```
void
CRB_interpret(CRB_Interpreter *interpreter)
{
    interpreter->execute_storage = MEM_open_storage(0);
    crb_add_std_fp(interpreter);
    crb_execute_statement_list(interpreter, NULL, interpreter->statement_list);
}
```

第一行准备了运行时要用的 MEM_Storage，第二行的函数 crb_add_std_fp() 注册了三个内部全局变量 STDIN、STDOUT 和 STDERR，然后通过 crb_execute_statement_list() 将解释器中保存的语句链表按顺序执行。

那么我们就来看一看 crb_execute_statement_list() 函数（代码清单 3-7）。

代码清单 3-7
crb_execute_statement_list()

```
StatementResult
crb_execute_statement_list(CRB_Interpreter *inter, LocalEnvironment
    *env, StatementList *list)
{
    StatementList *pos;
    StatementResult result;

    result.type = NORMAL_STATEMENT_RESULT;
    for (pos = list; pos; pos = pos->next) {
```

```
        result = execute_statement(inter, env, pos->statement);
        if (result.type != NORMAL_STATEMENT_RESULT)
            goto FUNC_END;
    }

 FUNC_END:
    return result;
}
```

即按照链表的顺序，调用 execute_statement()。

execute_statement() 则会根据不同的 Statement 类型执行不同的处理（代码清单 3-8）。

代码清单 3-8
execute_statement()

```
static StatementResult
execute_statement(CRB_Interpreter *inter, LocalEnvironment *env,
                  Statement *statement)
{
    StatementResult result;

    result.type = NORMAL_STATEMENT_RESULT;

    switch (statement->type) {
    case EXPRESSION_STATEMENT:
        crb_eval_expression(inter, env, statement->u.expression_s);
        break;
    case GLOBAL_STATEMENT:
        result = execute_global_statement(inter, env, statement);
        break;
    case IF_STATEMENT:
        result = execute_if_statement(inter, env, statement);
        break;
    case WHILE_STATEMENT:
        result = execute_while_statement(inter, env, statement);
        break;
     ⋮
    case STATEMENT_TYPE_COUNT_PLUS_1:   /* FALLTHRU */
    default:
        DBG_panic(("bad case...%d", statement->type));
    }

    return result;
}
```

execute_statement() 的第二个参数需要传递 LocalEnvironment 类型的结构体（指向其的指针）。这个结构体保存了当前运行中的函数的局部变量，

如果函数还没有运行，则传递 NULL。

execute_statement() 内部采用了 switch case 来区分条件处理。如果将其调用的函数全部进行分析的话有点浪费篇幅了，这里以 while 语句调用的 execute_while_statement() 为代表进行说明（代码清单 3-9，移除了错误检查等与核心功能无关的代码）：

代码清单 3-9
execute_while_statement()

```
static StatementResult
execute_while_statement(CRB_Interpreter *inter, LocalEnvironment
    *env, Statement *statement)
{
    StatementResult result;
    CRB_Value   cond;

    result.type = NORMAL_STATEMENT_RESULT;
    for (;;) {  /* 首先是一个无限循环 */
        /* 通过条件语句判别 */
        cond = crb_eval_expression(inter, env, statement->u.while_
            s.condition);
        /* 条件为真则结束循环 */
        if (!cond.u.boolean_value)
            break;

        /* 条件不为真则执行内部语句 */
        result = crb_execute_statement_list(inter, env,
                                           statement->u.while_s.block
                                           ->statement_list);

        /* break, continue, return 的处理 */
        if (result.type == RETURN_STATEMENT_RESULT) {
            break;
        } else if (result.type == BREAK_STATEMENT_RESULT) {
            result.type = NORMAL_STATEMENT_RESULT;
            break;
        }
    }

    return result;
}
```

if 语句或 while 语句都会包含一些内部的语句，这些内部语句被称为嵌套（nest）。在 crowbar 中，如果存在嵌套，则不会进行递归，而是转向运行嵌套内的语句。

事实上，实现上面的机制主要用到的是 break、continue、return 等，从

某种程度上来说用 goto 实现结构控制反而非常麻烦。

break、continue、return 等出现时，必须从递归的最深处强制返回上层。

break 或 continue 的作用是从最内层的循环中跳出，当然 break 或 continue 很可能会出现在很多层嵌套的 if 语句中，而无论其出现在哪里，正在进行的递归都必须从最底层一次性返回，这是不会改变的。

为了实现这一点，我们有一个“必杀技”可以使用——setjmp()/longjmp()*。这个必杀技还被应用到异常处理机制中，因此在 crowbar 中，返回值与结束状态会逐步返回。

* Ruby 的设计中也使用了这种必杀技。

execute_statement() 以及内部被调用的函数群 execute_XXX_statement()，返回值均为 StatementResult 的结构体。StatementResult 的定义如下：

```
typedef enum {
    NORMAL_STATEMENT_RESULT = 1,
    RETURN_STATEMENT_RESULT,
    BREAK_STATEMENT_RESULT,
    CONTINUE_STATEMENT_RESULT,
    STATEMENT_RESULT_TYPE_COUNT_PLUS_1
} StatementResultType;

typedef struct {
    StatementResultType type;
    union {
        CRB_Value       return_value;
    } u;
} StatementResult;
```

通常，type 会返回一个装入 NORMAL_STATEMENT_RESULT 的 StatementResult，而当执行 return、break、continue 时，则分别在 RETURN_STATEMENT_RESULT、BREAK_STATEMENT_RESULT、CONTINUE_STATEMENT_RESULT 装入对应的 StatementResult 并返回。此外，在代码清单 3-9 的 execute_while_statement() 等中会根据执行语句的返回值不同而所有不同，如果是 BREAK_STATEMENT_RESULT 或 RETURN_STATEMENT_RESULT 则会中断循环，而如果是 continue 的话，只会中断 crb_execute_statement_list() 中的运行。

如果是 return，那么不只要中断函数的运行，还要携带其返回值返回，此时会将返回值放入 StatementResult 的 return_value 中。

3.3.7 表达式求值——eval.c

表达式求值在 eval.c 中进行。

表达式求值，是在调用分析树进行递归下降分析后，通过对分析结果进行运算来实现的。其运算结果会装入 CRB_Value 结构体中并返回。关于 CRB_Value 的定义请参考 3.3.8 节。

语句的运行是通过 execute_statement() 中的 switch case 判断分支条件来实现的，而表达式求值则是通过 eval_expression() 来执行的（代码清单 3-10）。

代码清单 3-10
eval_expression()

```
 static CRB_Value
eval_expression(CRB_Interpreter *inter, LocalEnvironment *env,
                Expression *expr)
{
    CRB_Value   v;
    switch (expr->type) {
    case BOOLEAN_EXPRESSION:  /* 布尔型变量 */
        v = eval_boolean_expression(expr->u.boolean_value);
        break;
    case INT_EXPRESSION:   /* 整数型变量 */
        v = eval_int_expression(expr->u.int_value);
        break;
    case DOUBLE_EXPRESSION:  /* 实数型变量 */
        v = eval_double_expression(expr->u.double_value);
        break;
    case STRING_EXPRESSION:  /* 字符串变量 */
        v = eval_string_expression(inter, expr->u.string_value);
        break;
    case IDENTIFIER_EXPRESSION:  /* 变量 */
        v = eval_identifier_expression(inter, env, expr);
        break;
    case ASSIGN_EXPRESSION:  /* 赋值表达式 */
        v = eval_assign_expression(inter, env,
                                 expr->u.assign_expression.variable,
                                 expr->u.assign_expression.operand);
        break;
    /* 大部分二元运算符都整合在 eval_binary_expression() 中 */
    case ADD_EXPRESSION:        /* FALLTHRU */
    case SUB_EXPRESSION:        /* FALLTHRU */
    case MUL_EXPRESSION:        /* FALLTHRU */
    case DIV_EXPRESSION:        /* FALLTHRU */
    case MOD_EXPRESSION:        /* FALLTHRU */
```

```
    case EQ_EXPRESSION: /* FALLTHRU */
    case NE_EXPRESSION: /* FALLTHRU */
    case GT_EXPRESSION: /* FALLTHRU */
    case GE_EXPRESSION: /* FALLTHRU */
    case LT_EXPRESSION: /* FALLTHRU */
    case LE_EXPRESSION:
        v = crb_eval_binary_expression(inter, env,
                                       expr->type,
                                       expr->u.binary_expression.left,
                                       expr->u.binary_expression.right);
        break;
    /* 逻辑与，逻辑或 */
    case LOGICAL_AND_EXPRESSION:/* FALLTHRU */
    case LOGICAL_OR_EXPRESSION:
        v = eval_logical_and_or_expression(inter, env, expr->type,
                                        expr->u.binary_expression.left,
                                        expr->u.binary_expression.right);
        break;
    case MINUS_EXPRESSION: /* 单目取负运算符 */
        v = crb_eval_minus_expression(inter, env, expr->u.minus_expression);
        break;
    case FUNCTION_CALL_EXPRESSION:  /* 调用函数 */
        v = eval_function_call_expression(inter, env, expr);
        break;
    case NULL_EXPRESSION:  /* 常数 null */
        v = eval_null_expression();
        break;
    case EXPRESSION_TYPE_COUNT_PLUS_1:  /* FALLTHRU */
    default:
        DBG_panic(("bad case. type..%d\n", expr->type));
    }
    return v;
}
```

比如 Expression 结构体的 type 是 INT_EXPRESSION（整数的常数）时会调用 eval_int_expression()，其实现如下所示。最后在 CRB_Value 结构体中装入值并返回。

```
static CRB_Value
eval_int_expression(int int_value)
{
    CRB_Value   v;
    v.type = CRB_INT_VALUE;
    v.u.int_value = int_value;
    return v;
}
```

再比如分析树的加法节点（ADD_EXPRESSION）对于其左右项目会分别调用 eval_expression()，然后对其值进行加法运算，最后装入 CRB_Value 并返回。

上面看起来只有很简单的一句话，其实对于加法运算来说，左右项目的组合有以下几种情况：

1. 左边是整数，右边也是整数（整数相加，返回整数）
2. 左边是实数，右边也是实数（实数相加，返回实数）
3. 左边是整数，右边是实数（左边转换为实数，最后返回实数）
4. 左边是实数，右边是整数（右边转换为实数，最后返回实数）
5. 左边是字符串，右边也是字符串（返回连接后的字符串）
6. 左边是字符串，右边是整数（右边转换为字符串，返回连接后的字符串）
7. 左边是字符串，右边是实数（右边转换为字符串，返回连接后的字符串）
8. 左边是字符串，右边是布尔型（右边转换为内容是true或false的字符串，返回连接后的字符串）
9. 左边是字符串，右边是null（右边转换为内容是null的字符串，返回连接后的字符串）

对于 1~4 项来说，不只是加法，减法和乘法也都必须进行这样的类型转换 *，因此单独将评估加法表达式的函数独立出来并不是好的处理方式。crowbar 使用 eval_binary_expression() 函数对所有的二元运算符进行评估（代码清单 3-11），实际上这个函数中已经将同为整数、同为实数的运算单独划分为函数处理了，但代码整体还是很长（虽然有点长不过并不难，请读者尝试阅读一下）。这其中调用的子函数 eval_binary_int() 请参考代码清单 3-12。

* 在C语言中做除法运算，用整数除以整数商仍然为整数。而对于无类型语言来说，这种处理方式似乎不够好（crowbar 中商也为整数）。对于函数的参数来说，由于没有变量类型检查，因此如果向一个入口参数应为实数的函数传入整数值，Debug 时肯定会非常困难，并且容易产生 BUG[3]。

代码清单 3-11 eval_binary_expression()

```
 CRB_Value
crb_eval_binary_expression(CRB_Interpreter *inter, LocalEnvironment
    *env, ExpressionType operator, Expression *left, Expression *right)
{
    CRB_Value   left_val;
    CRB_Value   right_val;
    CRB_Value   result;

    left_val = eval_expression(inter, env, left);
    right_val = eval_expression(inter, env, right);

    if (left_val.type == CRB_INT_VALUE
        && right_val.type == CRB_INT_VALUE) {
        eval_binary_int(inter, operator,
                        left_val.u.int_value, right_val.u.int_value,
                        &result, left->line_number);
```

```
    } else if (left_val.type == CRB_DOUBLE_VALUE
               && right_val.type == CRB_DOUBLE_VALUE) {
        eval_binary_double(inter, operator,
                           left_val.u.double_value, right_val.u.double_value,
                           &result, left->line_number);
    } else if (left_val.type == CRB_INT_VALUE
               && right_val.type == CRB_DOUBLE_VALUE) {
        left_val.u.double_value = left_val.u.int_value;
        eval_binary_double(inter, operator,
                           left_val.u.double_value, right_val.u.double_value,
                           &result, left->line_number);
    } else if (left_val.type == CRB_DOUBLE_VALUE
               && right_val.type == CRB_INT_VALUE) {
        right_val.u.double_value = right_val.u.int_value;
        eval_binary_double(inter, operator,
                           left_val.u.double_value, right_val.u.double_value,
                           &result, left->line_number);
    } else if (left_val.type == CRB_BOOLEAN_VALUE
               && right_val.type == CRB_BOOLEAN_VALUE) {
        result.type = CRB_BOOLEAN_VALUE;
        result.u.boolean_value
            = eval_binary_boolean(inter, operator,
                                  left_val.u.boolean_value,
                                  right_val.u.boolean_value,
                                  left->line_number);
    } else if (left_val.type == CRB_STRING_VALUE
               && operator == ADD_EXPRESSION) {
        char   buf[LINE_BUF_SIZE];
        CRB_String *right_str;

        if (right_val.type == CRB_INT_VALUE) {
            sprintf(buf, "%d", right_val.u.int_value);
            right_str = crb_create_crowbar_string(inter, MEM_strdup(buf));
        } else if (right_val.type == CRB_DOUBLE_VALUE) {
            sprintf(buf, "%f", right_val.u.double_value);
            right_str = crb_create_crowbar_string(inter, MEM_strdup(buf));
        } else if (right_val.type == CRB_BOOLEAN_VALUE) {
            if (right_val.u.boolean_value) {
                right_str = crb_create_crowbar_string(inter,
                                                      MEM_strdup("true"));
            } else {
                right_str = crb_create_crowbar_string(inter,
                                                      MEM_strdup("false"));
            }
        } else if (right_val.type == CRB_STRING_VALUE) {
            right_str = right_val.u.string_value;
        } else if (right_val.type == CRB_NATIVE_POINTER_VALUE) {
```

```
            sprintf(buf, "(%s:%p)",
                    right_val.u.native_pointer.info->name,
                    right_val.u.native_pointer.pointer);
            right_str = crb_create_crowbar_string(inter, MEM_strdup(buf));
        } else if (right_val.type == CRB_NULL_VALUE) {
            right_str = crb_create_crowbar_string(inter, MEM_strdup("null"));
        }
        result.type = CRB_STRING_VALUE;
        result.u.string_value = chain_string(inter,
                                             left_val.u.string_value,
                                             right_str);
    } else if (left_val.type == CRB_STRING_VALUE
               && right_val.type == CRB_STRING_VALUE) {
        result.type = CRB_BOOLEAN_VALUE;
        result.u.boolean_value
            = eval_compare_string(operator, &left_val, &right_val,
                                  left->line_number);
    } else if (left_val.type == CRB_NULL_VALUE
               || right_val.type == CRB_NULL_VALUE) {
        result.type = CRB_BOOLEAN_VALUE;
        result.u.boolean_value
            = eval_binary_null(inter, operator, &left_val, &right_val,
                              left->line_number);
    } else {
        char *op_str = crb_get_operator_string(operator);
        crb_runtime_error(left->line_number, BAD_OPERAND_TYPE_ERR,
                          STRING_MESSAGE_ARGUMENT, "operator", op_str,
                          MESSAGE_ARGUMENT_END);
    }

    return result;
}
```

代码清单 3-12
eval_binary_int()

```
static void
eval_binary_int(CRB_Interpreter *inter, ExpressionType operator,
                int left, int right,
                CRB_Value *result, int line_number)
{
    if (dkc_is_math_operator(operator)) {
        result->type = CRB_INT_VALUE;
    } else if (dkc_is_compare_operator(operator)) {
        result->type = CRB_BOOLEAN_VALUE;
    } else {
        DBG_panic(("operator..%d\n", operator));
    }

    switch (operator) {
```

```
    case BOOLEAN_EXPRESSION:    /* FALLTHRU */
    case INT_EXPRESSION:        /* FALLTHRU */
    case DOUBLE_EXPRESSION:     /* FALLTHRU */
    case STRING_EXPRESSION:     /* FALLTHRU */
    case IDENTIFIER_EXPRESSION: /* FALLTHRU */
    case ASSIGN_EXPRESSION:
        DBG_panic(("bad case...%d", operator));
        break;
    case ADD_EXPRESSION:
        result->u.int_value = left + right;
        break;
    case SUB_EXPRESSION:
        result->u.int_value = left - right;
        break;
    case MUL_EXPRESSION:
        result->u.int_value = left * right;
        break;
    case DIV_EXPRESSION:
        result->u.int_value = left / right;
        break;
    case MOD_EXPRESSION:
        result->u.int_value = left % right;
        break;
    case LOGICAL_AND_EXPRESSION:        /* FALLTHRU */
    case LOGICAL_OR_EXPRESSION:
        DBG_panic(("bad case...%d", operator));
        break;
    case EQ_EXPRESSION:
        result->u.boolean_value = left == right;
        break;
    case NE_EXPRESSION:
        result->u.boolean_value = left != right;
        break;
    case GT_EXPRESSION:
        result->u.boolean_value = left > right;
        break;
    case GE_EXPRESSION:
        result->u.boolean_value = left >= right;
        break;
    case LT_EXPRESSION:
        result->u.boolean_value = left < right;
        break;
    case LE_EXPRESSION:
        result->u.boolean_value = left <= right;
        break;
    case MINUS_EXPRESSION:              /* FALLTHRU */
    case FUNCTION_CALL_EXPRESSION:      /* FALLTHRU */
    case NULL_EXPRESSION:               /* FALLTHRU */
```

```
        case EXPRESSION_TYPE_COUNT_PLUS_1:  /* FALLTHRU */
        default:
            DBG_panic(("bad case...%d", operator));
        }
    }
```

类似这样，在运行表达式求值时，会进行值的类型判定等很多处理，这也就是 crowbar 这样的无类型语言运行速度慢的原因之一 *。

* 对编译器进行优化的话，其实可以在编译阶段对值的类型进行判定。

在评估其他表达式时，比较重要的是调用函数。调用函数时会执行 `eval_function_call_expression()`（代码清单 3-13）。

代码清单 3-13 eval_function_call_expression()

```
static CRB_Value
eval_function_call_expression(CRB_Interpreter *inter, LocalEnvironment
        *env, Expression *expr)
{
        CRB_Value           value;
        FunctionDefinition  *func;

        char *identifier = expr->u.function_call_expression.identifier;

        func = crb_search_function(identifier);
        if (func == NULL) {
            crb_runtime_error(expr->line_number, FUNCTION_NOT_FOUND_ERR,
                              STRING_MESSAGE_ARGUMENT, "name", identifier,
                              MESSAGE_ARGUMENT_END);
        }
        switch (func->type) {
        case CROWBAR_FUNCTION_DEFINITION:
            value = call_crowbar_function(inter, env, expr, func);
            break;
        case NATIVE_FUNCTION_DEFINITION:
            value = call_native_function(inter, env, expr, func->u.native_f.proc);
            break;
        default:
            DBG_panic(("bad case..%d\n", func->type));
        }

        return value;
}
```

这个函数本身只负责将 crowbar 中的函数和 C 语言中的函数（内置函数）按条件区分处理。

如果是 crowbar 中的函数，会调用 `call_crowbar_function()`（代码清单 3-14）。

代码清单 3-14
call_crowbar_function()

```
 static CRB_Value
call_crowbar_function(CRB_Interpreter *inter, LocalEnvironment *env,
                      Expression *expr, FunctionDefinition *func)
{
    CRB_Value   value;
    StatementResult     result;
    ArgumentList        *arg_p;
    ParameterList       *param_p;
    LocalEnvironment    *local_env;

    /* 开辟空间用于存放被调用函数的局部变量 */
    local_env = alloc_local_environment();

    /* 对参数进行评估，并存放到局部变量中
       arg_p 指向函数调用的实参链表
       param_p 指向函数定义的形参链表 */
    for (arg_p = expr->u.function_call_expression.argument,
             param_p = func->u.crowbar_f.parameter;
         arg_p;
         arg_p = arg_p->next, param_p = param_p->next) {
        CRB_Value arg_val;

        if (param_p == NULL) {  /* param_p 被用尽：说明实参过多 */
            crb_runtime_error(expr->line_number, ARGUMENT_TOO_MANY_ERR,
                              MESSAGE_ARGUMENT_END);
        }
        arg_val = eval_expression(inter, env, arg_p->expression);
        crb_add_local_variable(local_env, param_p->name, &arg_val);
    }
    if (param_p) { /* param_p 剩余：说明实参数量不够 */
        crb_runtime_error(expr->line_number, ARGUMENT_TOO_FEW_ERR,
                          MESSAGE_ARGUMENT_END);
    }
    /* 运行函数内部语句 */
    result = crb_execute_statement_list(inter, local_env,
                                        func->u.crowbar_f.block
                                        ->statement_list);

    /* 如果 return 语句已经运行，则返回其返回值 */
    if (result.type == RETURN_STATEMENT_RESULT) {
        value = result.u.return_value;
    } else {
        value.type = CRB_NULL_VALUE;
    }
    dispose_local_environment(inter, local_env);

    return value;
}
```

crowbar 与 C 语言一样，是局部变量的生命周期，在函数被销毁时截止（局部变量生成的时机与 C 语言不同，是在变量被赋值时生成的）。因此在一个函数开始时，需要为这个函数准备一个运行环境，随着赋值开始注册新生成的局部变量，同时在函数结束后将其运行环境一同废弃。按照这个思路，我们为函数的运行环境准备了 LocalEnvironment 结构体。

LocalEnvironment 结构体的定义如下：

```
typedef struct {
    Variable    *variable;  /* 保存局部变量的链表 */
    GlobalVariableRef  *global_variable; /* 根据 global 语句生成的引用全局变量的链表 */
} LocalEnvironment;
```

Variable 是为了保存全局变量而使用的结构体（参考 3.3.1 节）。该结构体是通过链表构建的，链表内保存了函数内的局部变量。

函数的参数中包含了所有函数内要用到的局部变量，通过调用 call_crowbar_function() 中评估的 crb_add_local_variable() 函数将变量装入 LocalEnvironment。

全局变量在函数内被引用时，crowbar 中需要使用 global 语句进行声明（参考 3.1.4 节）。LocalEnvironment 结构体的 global_variable 成员中保存了 global 语句声明的指向全局变量的引用链表，其类型 GlobalVariableRef 的定义如下所示：

```
typedef struct GlobalVariableRef_tag {
    Variable    *variable;  /* 指向全局变量 */
    struct GlobalVariableRef_tag *next;
} GlobalVariableRef;
```

GlobalVariableRef 结构体在 global 语句运行时生成，同时被追加到 LocalEnvironment 结构体中。

3.3.8 值——CRB_Value

执行表达式求值的函数 eval_XXX_expression()，它的返回值类型为 CRB_Value。CRB_Value 类型的定义如下：

```
/* 类型的类别枚举 */
typedef enum {
```

```
    CRB_BOOLEAN_VALUE = 1,        /* 布尔型 */
    CRB_INT_VALUE,                /* 整数型 */
    CRB_DOUBLE_VALUE,             /* 实数型 */
    CRB_STRING_VALUE,             /* 字符串型 */
    CRB_NATIVE_POINTER_VALUE,     /* 原生指针型 */
    CRB_NULL_VALUE                /* NULL */
} CRB_ValueType;

typedef struct {
    CRB_ValueType       type;     /* 这个成员用于区别类型 */
    union {                       /* 实际的值保存在联合体中 */
        CRB_Boolean     boolean_value;
        int             int_value;
        double          double_value;
        CRB_String      *string_value;
        CRB_NativePointer       native_pointer;
    } u;
} CRB_Value;
```

CRB_Value 结构体用于保存 crowbar 中的每个值，并用枚举型区别值的类型，实际的值最终保存在联合体中。

对于无类型语言处理器来说，通常都需要像这样在值中保存值本身的类型。

3.3.9 原生指针型

原生指针型的值，通过 CRB_Value 结构体中的联合体成员 CRB_NativePointer 类型体现出来，其定义如下所示：

```
typedef struct {
    CRB_NativePointerInfo       *info;
    void                        *pointer;
} CRB_NativePointer;
```

pointer 成员是指向一些内部对象的指针。比如可以用原生指针型来做文件指针，此时指针实际会指向 FILE* 类型。为了可以容纳任何类型的指针，这里特意设置为 void*。

另一个成员 info 保存了原生指针型的信息，因此原生指针型本身就可以获得自己的类型。如果没有 info 成员只保存 void* 的话，当错误的类型传递给原生指针型时，内置函数将失去类型检查的方法，引起程序崩溃。作为 crowbar 的解释器，如果还会被 crowbar 程序的 BUG 弄崩溃就有点说不过去了。

info 成员的类型 CRB_NativePointerInfo 的定义如下所示：

```
typedef struct {
    char        *name;
} CRB_NativePointerInfo;
```

这里的 name 成员中，如果原生指针为文件指针时，成员值为 crowbar.lang.file 的字符串。

虽然经过上面的处理，可以对原生指针型进行类型检查，但这还不算是万全之策。比如为 FILE* 类型时，一个地方用 fclose() 关闭了文件指针，可能还会在另一个地方被使用。为了规避这种情况，原生指针型不应该直接指向 FILE*，而是需要经过一个第三方对象。不过当前版本的 crowbar 对第三方对象的释放处理仍然存在问题。之后的版本会引入标记 - 清除 GC，届时这个问题会一并解决，请参考 4.4.5 节。

3.3.10 变量

我们先后提到了局部变量与全局变量，那么接下来，我再对变量做一些介绍。

变量名在表达式中出现时，通过 eval_expression()（参考代码清单 3-10）调用 eval_identifier_expression()（代码清单 3-15）。

代码清单 3-15 eval_identifier_expression()

```
static CRB_Value
eval_identifier_expression(CRB_Interpreter *inter,
                           LocalEnvironment *env, Expression *expr)
{
    CRB_Value   v;
    Variable    *vp;

    /* 首先查找局部变量 */
    vp = crb_search_local_variable(env, expr->u.identifier);
    if (vp != NULL) {
        v = vp->value;
    } else {
        /* 如果没有找到，则通过 CRB_Interpreter 或 LocalEvironment 连接
           GlobalVariableRef，在其中查找全局变量 */
        vp = search_global_variable_from_env(inter, env, expr->u.identifier);
        if (vp != NULL) {
            v = vp->value;
        } else {
            /* 仍然没有找到则报错 */
```

```
            crb_runtime_error(expr->line_number, VARIABLE_NOT_FOUND_ERR,
                              STRING_MESSAGE_ARGUMENT,
                              "name", expr->u.identifier,
                              MESSAGE_ARGUMENT_END);
        }
    }
    refer_if_string(&v); /* 这里下文会详述 */

    return v;
}
```

crb_search_local_variable() 会从 LocalEnvironment 中查找局部变量。另外，crb_search_global_variable() 的第二个参数 LocalEnvironment 为空的话（即在顶层结构中），会从 CRB_Interpreter 中查找。如果两者都没有找到，则从第二个参数 LocalEnvironment 连接的 GlobalVariableRef 中查找全局变量。

变量赋值时的处理通过 eval_assign_expression() 进行（代码清单 3-16）。

代码清单 3-16
eval_assign_expression()

```
static CRB_Value
eval_assign_expression(CRB_Interpreter *inter, LocalEnvironment *env,
                       char *identifier, Expression *expression)
{
    CRB_Value   v;
    Variable    *left;

    /* 首先评估右边 */
    v = eval_expression(inter, env, expression);

    /* 查找局部变量 */
    left = crb_search_local_variable(env, identifier);
    if (left == NULL) {
        /* 没有找到则查找全局变量 */
        left = search_global_variable_from_env(inter, env, identifier);
    }
    if (left != NULL) {  /* 找到变量 */
        release_if_string(&left->value); /* 本行后文会详述 */
        left->value = v;  /* 在这里赋值 */
        refer_if_string(&v); /* 本行后文会详述 */
    } else { /* 因为没有变量，所以新生成 */
        if (env != NULL) {
            /* 函数内注册局部变量 */
            crb_add_local_variable(env, identifier, &v);
        } else {
            /* 函数外注册全局变量 */
```

```
            CRB_add_global_variable(inter, identifier, &v);
        }
        refer_if_string(&v); /* 本行后文详述 */
    }

    return v;
}
```

赋值处理的流程已经写入注释，其中后文详述的部分会在之后的章节中说明。

当前的语法规则中，赋值表达式的左边必须保证为变量名。如果之后可以使用数组等新语法元素，可能会有下面这样的赋值：

```
a[b[func()]] = 10;
```

左边可能为非常复杂的表达式，因此还需要进一步考虑。

3.3.11 字符串与垃圾回收机制——string_pool.c

如上文所写，字符串类型通过 + 连接的同时，需要配合某种**垃圾回收机制**(garbage collector，GC)。当前版本的 crowbar 是通过**引用计数**这种原始的垃圾回收机制实现的。

引用计数，即通过管理指向某些对象（这里是字符串）的指针数量，在计数为零时将其释放的回收机制。

crowbar 的字符串保存在 `CRB_String` 结构体中。

```
typedef struct CRB_String_tag {
    int         ref_count;  /* 引用计数 */
    char        *string;    /* 字符串本身 */
    CRB_Boolean is_literal; /* 是否为字面常量 */
}   CRB_String;
```

这个结构体的 `string` 最终指向存放字符串的区域。

`CRB_String` 会在下列的时机中生成，生成时引用计数被置为 1。

1. 字符串字面常量被评估时。
2. 通过+运算符生成新的字符串时。
3. 通过+运算符将整数型或实数型转换为字符串类型时。
4. 通过`fget()`函数读入文件时。

引用计数为零时回收。此时字符串本身（即 `string` 成员的指向）一般

来说也需要被释放，但如果字符串为字面常量（literal，即被 "" 包裹的出现在 crowbar 代码中的字符串）则不释放。字面常量不是通过 MEM_malloc() 而是通过 MEM_storage_malloc() 在 interpreter_storage 中开辟的。区别字符串是否为字面常量，通过其成员 is_literal 即可，也就是说，字符串字面常量所对应的 CRB_String，生成于赋值表达式的字符串字面常量被评估时，但是其指向的字符串本身，还会在分析树构建时被重复使用。

引用 CRB_String 的指针会存在于以下几处地方：

1. 局部变量
2. 全局变量
3. 函数的参数
4. eval.c中的局部变量、参数、返回值等，eval.c运行时的C栈

那么只需要在这些指针被引用时将引用计数自增，引用解除时将引用计数自减即可。

因此，在程序中需要在以下的时机对引用计数进行自增处理（条目编号与上文 CRB_String 生成的编号一一对应）。

1. 局部变量被赋值为字符串时。
2. 全局变量被赋值为字符串时。
3. 函数的参数被赋值为字符串时。这里有个例外，如果入口参数为实参时，其表达式求值结束时会伴随一次自减，会与本应进行的计数自增两相抵消。
4. 字符串的变量被评估时。除此之外，当字符串出现在表达式中时，会新生成CRB_String。

而程序中还需要在以下的时机对引用计数进行自减处理。

1. 存放字符串的局部变量被复写时，以及退出函数，局部变量被释放时。
2. 存放字符串的全局变量被复写时，以及程序运行完毕，全局变量被释放时。
3. 从内置函数退出，释放入口参数时（如果是crowbar函数，入口参数是作为局部变量处理的，此时的处理参考上面的条件1）。
4. 返回字符串的表达式会对其进行表达式/语句评估，评估处理结束时。通过+运算符对字符串进行连接时。通过比较运算符比较字符串后，两边的字符串需要释放时。

在实际处理中，引用计数的自增通过 refer_if_string() 函数实现，其自减则通过 release_if_string() 函数（类似这种函数名中有 if_string 的函数，仅限于对象为字符串时才会进行处理）实现。参考代码清单 3-16 可以明显看到：

- 当变量被复写时，原来变量中字符串的引用计数会自减；
- 当变量被赋值为字符串时，该字符串的引用计数会自增。

再举一个复杂一点的例子，比如有如下语句时（hoge()函数的返回值为字符串类型）：

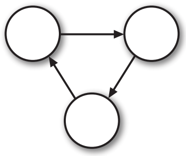
```
a = b = c = hoge( "piyo");
```

首先，在评估 piyo 时生成 CRB_String，引用计数被置为 1。将其传递给函数时，由于入口参数为实参，所以会与本应进行的计数自增相抵消，引用计数仍然保持为 1 不变。赋值给 c 后计数为 2，赋值给 b 后计数为 3，最后赋值给 a 后计数为 4。同时，由于对表达式语句的评估结束，又会进行一次自减，所以计数为 3。通过 a、b、c3 个变量的引用，应用计数最终为 3。

但是引用计数这种垃圾回收机制存在一个致命的缺陷，那就是无法释放**循环引用**。

循环引用如图 3-5 所示，即几个对象之间相互进行引用。

图 3-5
循环引用



这种状态下，所有对象的引用计数均为 1，因此通过引用计数的方式进行垃圾回收显然是行不通的。其实，如果能控制局部变量和全局变量令其无法引用的话，这种循环引用本来是不应该出现的。也就是说，以引用计数方式进行垃圾回收时，循环引用会引起内存泄漏。

当前的 crowbar 中，垃圾回收的对象只有字符串一种，而字符串是无法引用其他对象的，所以还不存在循环引用的问题。但是当下几乎所有的实用编程语言，都可以用一个对象保存指向另一个对象的引用，因此如果想扩展并支持这样的功能的话，引用计数式的垃圾回收机制就必须做出改进。

3.3.12 编译与运行

如 1.6.2 节所写，本书中所涉及的代码都可以在以下 URL 下载：

```
http://avnpc.com/pages/devlang#download
```



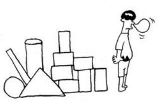
将下载的文件解压缩后，进入该文件夹并运行 make* 即可生成执行文件。

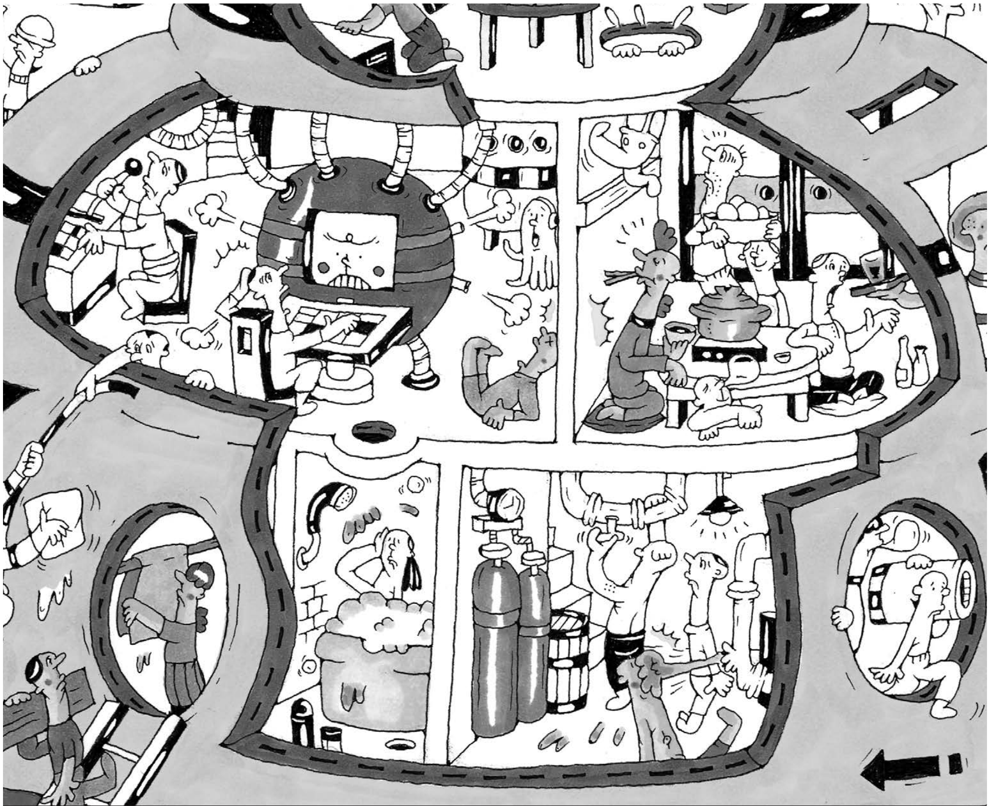
本书涉及的程序所需的包也附加在 Makefile 中了，只是在 Windows 所用的 Makefile 中，C 编译器名称已经设置为了 gcc，make 名称也设置为 gmake 了（参考 1.6.1 节）。如果你的系统环境不一样的话，请根据实际情况作适当调整。

解压的文件夹中还放入了一个 test.crb 的示例代码，可以通过下面的指令运行：

```
% crowbar test.crb
```

* make 是 UNIX 下经典的编译 / 自动化构建工具，而在集成开发环境中，很多 IDE 都可以直接从菜单中点击 Build 按钮进行编译。make 指令中，编译 / 链接等具体步骤都会写在 Makefile 文件中。





第 4 章

数组和标记 – 清除垃圾回收器



4.1 crowbar ver.0.2

crowbar book_ver.0.1 不能使用数组，会让用户感觉不太实用，因此在 book_ver.0.2 中我们将引入数组的概念。

4.1.1 crowbar 的数组

在 crowbar ver.0.2 中，可以像代码清单 4-1 那样使用数组。

代码清单 4-1 数组

```
# 创建数组
a={1,2,3,4,5,6,7,8};

# 显示数组
for(i = 0; i < a.size(); i++){
    print("(" + a[i] + ")");
}

print("\n");

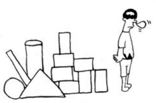
# 创建九九乘法表
a99 = new_array(9,9);
for(i = 0; i < 9; i++){
    for(j = 0; j < 9; j++){
        a99[i][j] = (i+1) * (j+1);
    }
}

# 显示九九乘法表
for(i = 0; i < a99.size(); i = i + 1){
    for(j = 0; j < a99[i].size(); j = j + 1){
        print("[" + a99[i][j] + "]");
    }

    print("\n");
}

# 附带：显示字符串长度
print("len.." + "abc".length() + "\n");
```

与 C 语言不同，在大多数脚本语言中，数组（或者列表）都可以用字面量表示。例如 Perl 中，像这样：



```
(1, 2, 3)
```

就可以创建一个由 1、2、3 组成的数组（列表）。

Ruby 和 Python 是这样的：

```
[1, 2, 3]
```

而 Tcl 是这样的：

```
{1, 2, 3}
```

语言不同，数组的定义方式也不相同。我总觉得 crowbar 与 C 语言比较相似，既然在 C 中数组用 { } 初始化，那么在 crowbar 中也使用 { } 吧。

```
{1, 2, 3}
```

用这种方式来创建数组。同理，

```
a = {1, 2, 3};
```

就可以把数组赋值给变量。（如果这么写的话）可以像下面这样直接用下标指定元素。

```
{1, 2, 3}[1]
```

这个表达式的值是整数型的 2。

另外，数组的元素中可以包含数组或其他所有数据类型，并且每个元素都可以包含不同类型的数据。

```
a = {true, 1, "abc", {5,10.0}};
```

这样一来，在 a 中包含了布尔型、整数型、字符串和数组四个类型的元素，其中第 4 个元素包含了数组。`a[3]` 返回一个数组，`a[3][1]` 返回 10.0。如上所述，crowbar 中虽然不存在多维数组，但实际上可以用“数组的数组”方式来实现。

4.1.2 访问数组元素

从上一节给出的例子中我们不难看出，用 `[]` 的方式可以访问数组元素。当然，下标是从 0 开始的。

```
b = a[i];
```

这个语句把数组 a 的第 i 个元素赋值给变量 b。

```
a[i] = 5;
```

上面这个的语句将整数 5 赋值给 a 的第 i 个元素，但如果 a 不是数组类型的话，运行时就会出现错误。还有，[] 中必须是整数类型。

crowbar 的数组不支持自动扩展。在 Perl 等有些语言里，只要使用像 a[100] = 10; 这样的语句就可以让数组自动扩展，与数组当前的大小无关。在把所有数组都视为关联数组（字符串等也可以作为下标）的语言（如 JavaScript）中，也可以随时随地给 a[100] 赋值。但是，这种语言的设计方式在我看来迟早会出现 bug。在 crowbar 里，如果 a 的元素数是 5 的话，无论是给 a[10] 赋值还是引用 a[10] 元素，都会引发运行时的错误。

4.1.3 数组是一种引用类型

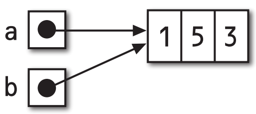
crowbar 的数组是一种引用类型。什么是引用类型呢？说白了，就是指向原始值的变量类型，其实就是指针。与 C 语言不同的是，crowbar 的数组类型不会发生访问错误内存地址的情况。

一个数组赋值给变量 a 时，实际上 a 保存的是“指向”这个数组的值，将这个值赋给其他变量的话，两个变量就指向了同一个数组。因此，下面这段程序会输出 a[1]..5。

```
a = {1, 2, 3};
b = a;
b[1] = 5;
print("a[1].." + a[1] + "\n");
```

把 a 赋值给 b，这样一来变量 a 和变量 b 就和图 4-1 一样，同时指向同一个数组。

图 4-1
两个变量引用同一个数组



补充知识　“数组的数组”和多维数组

前面提到过，crowbar 语言中虽然没有多维数组，但有了“数组的数组”基本上就

可以实现用多维数组做的事情了。

如果要在 crowbar 中引入多维数组的话，就要用下面这种形式引用元素了。

```
a[3, 4]
```

这种多维数组有一个不方便的地方，就是不能单独取出数组中的一部分。

举个例子，每天的销售额以月为单位存在数组中，如果想指定月份和日期取出某一日的销售额，要写成下面这样：

```
#取11月15日的销售额（month和day都从0开始）
uriage[10][14]
```

如果想要定义一个函数来计算某个月的总营业额，要写成下面这样，只把某个月的数组作为参数传递给这个函数。

```
#接受一个数组并计算合计值的函数（以11月的销售额为例）
calc_sum(uriage[10]);
```

上面的例子，数组的数组可以做到，但多维数组就不行了（特别是当calc_sum()为通用函数的时候）。

难道说多维数组就一点优点都没有吗？也不能这么说。由于在 crowbar 语言中数组是引用类型，因此数组的数组会像图 4-2 中那样分配内存，如果是多维数组的话也许会像图 4-3 那样。根据 malloc() 函数的机制，申请内存时多少都需要一些管理空间 *，申请多个不连续的空间会加重 GC 的负担。总而言之，多维数组的运行效率有可能比数组的数组要高一些 *。

* 不考虑 crowbar 的 MEM 实际会占用更大的管理空间。

* C 语言中只有“数组的数组”，数组也不属于引用类型，在这里道理是一样的。

图 4-2
“数组的数组”的内存结构

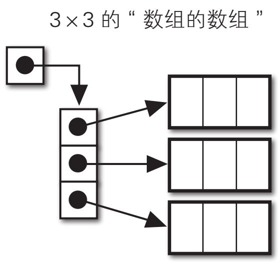
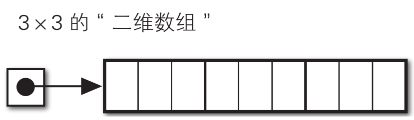


图 4-3
“多维数组”的内存结构



另外，使用数组的数组可以改变某个特定子数组的长度或者将其设置为 null。当然，要说它的方便性还能举出很多例子来，但是也有不需要它的时候，比如一个五子棋的棋盘，已经定好了是 8×8，这种情况下用多维数组来表示会更明确。

C#、D、Ada 等语言能够同时支持数组的数组和多维数组。

4.1.4 为数组添加元素

crowbar 的数组不能简单地使用赋值语句进行自动扩展，而是需要显式地添加元素来扩展数组。下面的代码在数组的末尾追加了一个元素。

```
a = {1, 2, 3};
a.add(4);
```

这样一来，a 指向的数组就变成了 {1， 2， 3， 4}。

根据实际情况，我们希望有些数组一次就生成指定的大小。比如，想要管理 40 名学生的身高，用索引值来表示学号，但是生成数据的顺序是随机的。像这种情况最好从一开始就预先生成一个 40 个元素的数组。

我觉得还是不要过多地摆弄语法，使用原生函数比较好。原生函数 new_array() 可以生成一个指定大小的数组。

```
a = new_array(40);
```

初始化状态下，所有的元素都是 null。

给这个函数传入多个参数，也可以生成多维数组（数组的数组）。

```
a = new_array(5, 10);
```

上例创建了一个元素最多到 a[4][9] 的数组。

4.1.5 增加调用（伪）方法的功能

前面的章节中，使用了一种方法调用式的语法结构为数组添加元素。

```
a.add(3);
```

为了支持这种方法调用，我们修改了语法结构，也为字符串增加了 length() 方法。

4.1.6 其他细节

在变更左值的处理方式的同时，我们顺便引入了自增和自减运算符。i++ 时 i 会增加 1，-- 时同理。在 crowbar 中，自增和自减运算符只能放在后面，不



支持前置的 ++ 和 --。

C 语言中自增和自减运算符前置和后置的含义不同。人们很难读懂在一行里面写满了表达式的代码，所以我觉得自增和自减最好还是独占一行，这样一来前置和后置的含义就相同了，不论写在哪边都可以。就我个人而言，一般习惯写在后面。

也许有人会问了，在一行里只能写一个自增和自减的话，为什么它们非要是表达式呢？变成语句不是更好吗？如果是这样的话，for 语句的第三个表达式就没法使用 i++ 了，所以它们还是用作表达式吧。

4.2 制作标注–清除 GC

crowbar book_ver.0.1 中采用的引用计数 GC 存在不能释放循环引用的问题，于是在 book_ver.0.2 中将实现一个标记 – 清除 GC。

4.2.1 引用数据类型的结构

在讨论 GC 的话题之前，先说明一下 crowbar ver.0.2 中引用数据类型的处理方式。

crowbar 中有以下两种引用数据类型：

- 数组
- 字符串

尤其是字符串，它是一个不允许改变内容（immutable）的对象，用户没有必要意识到它是一个引用（请参考 4.2.2 节的补充知识）。

crowbar 的所有值都保存在 CRB_Value 中。在 crowbar book_ver.0.1 里，CRB_Value 直接保存着指向 CRB_String 的指针。从现在开始，将增加新的 CRB_Object 类型用来统一处理字符串和数组，在 CRB_Value 中将保存指向 CRB_Object 的引用。

```
typedef struct{
    CRB_ValueType   type;
    union{
        CRB_Boolean boolean_value;
```

```
        int         int_value;
        double      double_value;
        CRB_NativePointer   native_pointer;
        CRB_Object  *object;/* 这个是新增的 */
    } u;
} CRB_Value;
```

用 CRB_Object 内的联合体来保存 CRB_String 以及为了这次数组而引入的类型 CRB_Array。

```
typedef enum{
    ARRAY_OBJECT = 1,
    STRING_OBJECT,
    OBJECT_TYPE_COUNT_PLUS_1
} ObjectType;

struct CRB_Object_tag{
    ObjectType type;
    unsigned int    marked:1;
    union{
        CRB_Array array;
        CRB_String string;
    } u;
    struct CRB_Object_tag *prev;
    struct CRB_Object_tag *next;
};
```

虽然通过 CRB_Value 可以明确地区分出数据类型，但是为了在 GC 的时候仅通过 CRB_Object 就可以能够辨别数据类型，必须在 CRB_Object 中加入一个 ObjectType 枚举类型的成员。

CRB_Object 的成员 marked 作为一个标记对象用的标识符，将用于后面要谈到的标记 – 清除 GC。至于它的数据类型，由于 1 个比特就足够了，因此选择了位域（bit field）。

说起位域这个功能，其实在 C 语言中不太会用到。即使在现在的自由软件中，应该也有不少自己进行位运算并为 int 变量附加各种标识的情况。但我认为，时至今日人们已无需刻意回避位域这个话题了。不过，如果从富翁式编程的角度出发，就算是 1 个比特的标志位，可能也要给它分配一个 CRB_Boolean 型。

CRB_Object 的联合体中有 CRB_Array 或 CRB_String，它们的定义如下：

```
struct CRB_Array_tag {
    int         size;/* 显示用的元素数 */
```

```
    int         alloc_size;/* 实际占用的元素数 */
    CRB_Value   *array;/* 数组元素（数组长度可变）*/
};

struct CRB_String_tag {
    CRB_Boolean is_literal;
    char        *string;
};
```

为了提高效率，数组在添加元素时会多扩展一些空间，所以在 size 属性之外还保存了 alloc_size 属性。

4.2.2 标记 – 清除 GC

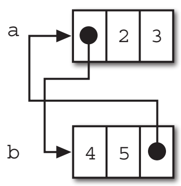
前面说过，book_ver.0.1 中使用的引用计数 GC 不能释放循环引用。在 book_ver.0.1 里，GC 对象只是字符串，并不会因为不能释放循环引用而出现问题。但是有了数组的数组，这样的方式在 book_ver.0.2 中就有可能引起问题，比如：

```
a = {1, 2, 3};
b = {4, 5, a};
a[0] = b;
```

这样一段代码就形成了图 4-4 中描述的循环引用。

发生循环引用时，即使它们整体上不再被引用，引用计数器此时也不为 0。这就是所谓的内存泄漏。

图 4-4
循环引用



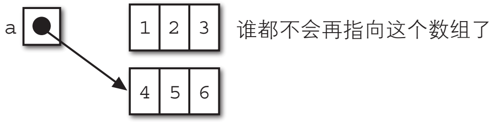
于是我们抛弃掉引用计数 GC，引入标记 – 清除 GC。本来所谓的垃圾回收（GC）就是要自动释放不使用的对象占据的内存空间的机制。但是，怎么去定义“不使用的对象”呢？其实就是“绝对不会被引用到的对象”。比如：

```
a = {1, 2, 3};
a = {2, 3, 4};
```

这样一段代码在执行了第二行的赋值语句后，{1, 2, 3} 这个数组就不再被引

用了，即成为了 GC 的对象（如图 4-5 所示）。

图 4-5
对象成为 GC 目标的例子



所谓标记－清除 GC 是一种直接实现定义“不使用的对象”的 GC 算法。这个算法有以下几个原则。

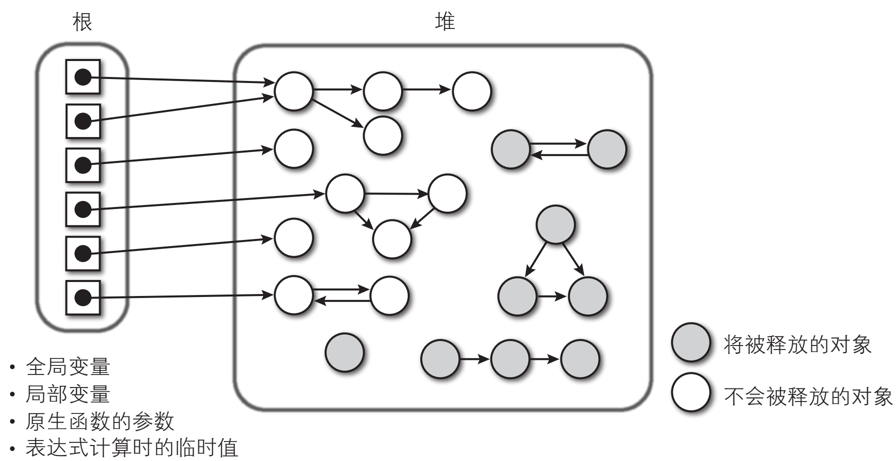
1. 从变量之类的“引用起点”开始，追溯所有能引用到的对象并标记。（标注阶段）
2. 将没有被标记的对象全部释放。（清除阶段）

crowbar 里能够成为“引用起点”的有以下几处，我们称这些“引用起点”为**根**（root 或 roots）。

1. 全局变量
2. 局部变量。包含用crowbar语言记述的函数的形式参数。
3. 原生函数的形式参数。
4. 表达式计算时的临时引用。

只有从这些根能够追溯到的对象才可以存留下来，其余的对象将被视作 GC 的目标被释放。

图 4-6
成为 GC 目标的对象



其实这些根中最难处理的就是表达式计算时的临时引用。比如有个表达式 `"abc" + "def" + "ghi"`,crowbar 的解释器首先会计算 `"abc" + "def"`,

生成字符串 "abcdef"*。这个字符串的存在是必要的，但不论是全局变量还是局部变量或者原生函数的形式参数都不会引用它，这样一来临时变量就会被 GC 丢弃，情况就会变得很糟。也许有人会想：“在这种关键时刻不要让 GC 启动不就行了？”下面这个例子就阐述了 crowbar 在此时要启动 GC 的原因。

* 现在的 crowbar 并没有在编译时处理多个字符串字面量的加法运算（虽然可以这么做），所以在这个例子中，使用字符串字面量进行加法运算的效果与使用字符串变量相同，这里并不是为了说明变量和字面量的差异。

```
print("abc" + "def" + long_long_function());
```

但本例中的 long_long_function() 在执行过程中不进行 GC 是不现实的，所以在表达式计算的过程中应该允许启动 GC。

crowbar 中 GC 的启动时机将在后面的章节介绍，大概的原则就是，在新分配内存空间的时候有可能会启动。

补充知识　引用和不可变类

在 crowbar 中，整数和实数类型的变量都是把值直接存在 CRB_Value 中，但是字符串和数组类型在 CRB_Value 中只保存引用。举例来说，Java 与 crowbar 相同，在 Java 中像 int 或者 double 这样的类型叫作**原始类型**（primitive type），而像字符串或者数组这样的类型叫作**引用类型**（reference type）。

有人可能会说，存在两个种类的数据类型有违编程之美，可是在 crowbar 中不必认为字符串是一种引用类型。至于数组，只有像下面这段代码编写出的数组才是一种引用类型。

```
a = {1, 2, 3};
b = a;
a[1] = 10; ←这行代码也可替换为b[1]，效果相同
```

如果是字符串的话，就没办法改变内容了（这样的数据类型称为不可变的数据类型）。比如用 + 运算符连接字符串，就会得到一个新的字符串对象，之前[①]字符串对象的内容不会发生改变。

```
a = "abc";
b = a;
a = a + "d"; ←a会变为"abcd"，b还是"abc"
```

另外，crowbar 中如果使用 == 比较字符串的话，不是进行引用之间的比较，而是比较字符串的内容。Java 在这种情况下比较的就是引用，难不成是特意让人感觉到字符串是一种引用？ *

* 也不知道 Java 这样的设计方便性在哪里。可能是会提升 intern() 的效率吧。

按照刚才的思路，可能只能让字符串看起来像是一个原始类型，数组仍然是引用类型。但这样的解释并不能让刚才说“存在两个种类的数据类型有违编程之美”的人满意吧。

① 即参与运算。——译者注

那么不妨逆向思考一下，可以把整数类型和实数类型都看作是引用类型。这样一来，不可变的引用类型和原始类型就看不出区别了。如果不考虑实现问题，可以勉强把整数和实数看作是不可变的引用类型。

不知道上面的解释能否使那些认为“存在两个种类的数据类型有违编程之美”的人满意，不过，在给编程新手讲这个问题时，这种解释是否行得通，我觉得就另当别论了。

现在的编程入门书，基本上都把变量解释为“像是用来放值的盒子”（我曾经也这样写过）。“盒子说”是用来解释原始类型的，如果要说明“一切都是引用”，就不得不引入其他说法（也许是“名片说”？）了。

“盒子说”也不是毫无问题的（b = a; 的时候，a 的内容转移到了 b 里面，那 a 不是应该变成空的了吗？）。大多数新手好像很难理解引用的概念。另外，关于“存在两个种类的数据类型有违编程之美”这种说法，新手也不会太在意统一性之类的事情（对编程语言有了一定程度的了解后才会在意）。以我的经验来看，如果是以教会新手编程为目的，与其让他们知道统一性之类的道理，不如用盒子说来说明会更加容易。总而言之，写代码才是对编程语言的学习最有帮助的。

4.2.3 crowbar 栈

让我们继续“表达式计算时的临时引用很难处理”这个话题。

如果你要问在哪儿出现了“表达式计算时的临时引用”，其实 crowbar book_ver.0.1 里面的 eval.c 中各函数的局部变量就是。

比如字符串连接时，`crb_eval_binary_expression()` 函数会按照以下顺序执行（具体代码省略）。

```
CRB_Value left_val;
CRB_Value right_val;
CRB_Value result;

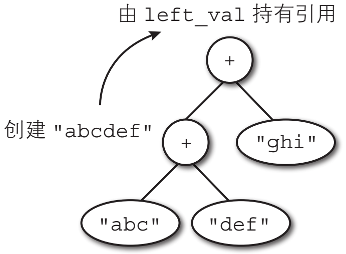
/* 计算左边的值 */
left_val = eval_expression(inter, env, left);
/* 计算右边的值 */
right_val = eval_expression(inter, env, right);

（中间省略）
     result.type = CRT_STRING_VALUE;
     /* chain_string 执行时，GC 可能会启动 */
     result.u.string_value = chain_string(inter,
                                          left_val.u.string_value,
                                          right_str);
```

```
（中间省略）
return result;
```

像"abc" + "def" + "ghi"这样的表达式，它的分析树如图4-7所示。因为是从最深层次开始计算，left_val会暂时持有指向字符串"abcedf"的引用，然后在调用chain_string()函数时，由于会申请内存空间，因此也有可能启动GC（关于GC启动的时机，我将在4.3.2节介绍）。此时，虽然会标记left_val指向的对象（chain_string()会被更深的层级调用），但是从GC的角度，是看不到只作为局部变量的left_val的。

图4-7
字符串连接的分析树



想要解决这个问题倒是有一个办法，就是扫描C的局部变量内存区域（Ruby使用的方法）。C语言的局部变量通常分配在栈上，所以只要在表达式求值开始时记录下相关局部变量的地址，并且在GC的时候再记录一次相关局部变量的地址，就足以保证在这期间分配的局部变量肯定在这两处地址中（假设没有被优化分配到寄存器上的话）。这样，扫描C的所有局部变量区域，若能发现形似引用对象指针的地方，就以此为起点开始标记。

只是这个方法有如下缺点，让我不太想用它。

- （虽然说移植性相当高，但是）依赖C语言的实现，有违编程之美。
- 因为不知道是栈中的哪些部分引用的对象，所以不得不采取“把看上去像对象引用的全部当做对象引用来处理”的方法。如果要处理的区域不是引用对象，就会发生内存泄漏。顺便说一句，这种GC叫作保守式GC。

就内存泄漏而言，我觉得（从Ruby应用的情况来看）在应用上不成问题*。虽说这是C语言，但要让我胡乱地将指针作为地址处理也是不可以的。

crowbar到底要怎么处理呢？既然C语言的栈这么不好用，那么全部用独立的栈管理不就好了吗？对此，把“表达式运算时的临时引用”放在独立的栈（CRB_Value数组）中，GC把这个栈中可以引用到的对象标记起来就可以了。

* 这种内存溢出会随着运行时间慢慢泄漏。这种性质的内存泄漏，在有些程序中会非常致命，这种类型的保守式GC中，泄露的对象数量与当时栈的长度成一定比例，而且这个比例几乎是固定的。

用 Stack 结构体来表示栈。

```
typedef struct {
    int         stack_alloc_size;
    int         stack_pointer;
    CRB_Value   *stack;
} Stack;
```

CRB_Interpreter 持有这个结构体（只有 CRB_Interpreter 会持有 Stack 结构体，这么做只是为了划分空间）。在这个栈中（以后就称为 crowbar 栈吧），与是否包含对象的引用无关，它会把表达式运算时产生的所有值都保存起来。

在此之前，所有的 eval.c 运算函数都将运算的结果值作为返回值，而 ver.0.2 以后的版本都将由栈返回。因此，之前使用过的 eval_xxx_expression() 系列函数的返回值也从 CRB_Value 变成 void 了。

例如，有如下表达式：

```
"1+2*3.." + (1 + 2 * 3);
```

会形成如图 4-8 这样的分析树，表达式运算时栈的变化如图 4-9 所示。

图 4-8
表达式的分析树

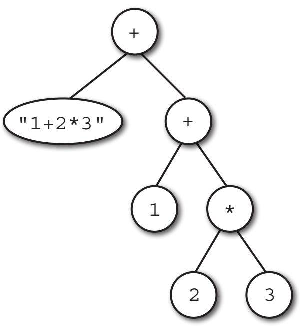
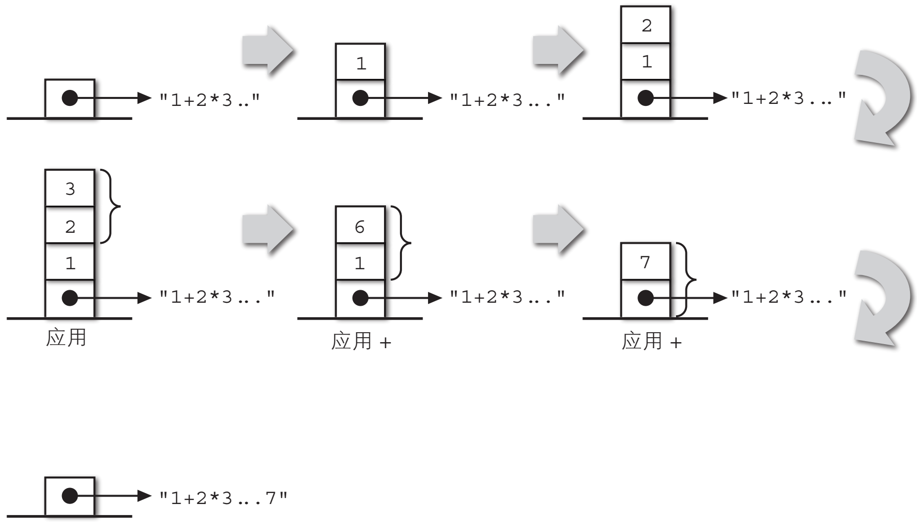


图 4-9
表达式运算过程中栈的变化



总而言之，就是从栈中获取了 * 和 + 运算符的操作数，并将运算结果的值留在了栈上。这种做法与 JVM 栈运行字节码时的栈动作相同。如此一来，我们就向字节码解释器又迈进了一步。

本书的后半部分将介绍字节码解释器型语言的制作方法。

4.2.4 其他根

接下来，将要说明除了“表达式运算时的临时引用”之外的三种根。

1. 全局变量

全局变量可以在 `CRB_Interpreter` 中以链表的形式追溯，很容易以此为起点进行标记。

2. 局部变量

在 `LocalEnvironment` 结构体中持有局部变量，并作为参数传递。

为了使 GC 能够全部标记当前生效的局部变量，并且在任何时候都能追踪到这些局部变量，在 `LocalEnvironment` 中增加了成员 `next`，使之成为链表（因为这个版本的结构体都写在 CRB_dev.h 中，所以加上 `CRB_` 作为前缀。请参考 4.4.4 节）。

```
struct CRB_LocalEnvironment_tag {
    Variable              *variable;
    GlobalVariableRef     *global_variable;
    RefInNativeFunc       *ref_in_native_method;/* 之后会讲到 */
    struct CRB_LocalEnvironment_tag *next;/* 新增加 */
};
```

链表的顶端由 CRB_Interpreter 持有。这里的“顶端”指的是最后被调用的函数的 CRB_LocalEnvironment。

```
struct CRB_Interpreter_tag {
    （省略）
    CRB_LocalEnvironment    *top_environment;
};
```

值得一提的是，当前的 crowbar 版本在搜索局部变量的时候，会从最近一次作为参数传递的 LocalEnvironment 开始搜索，顺着 next 从函数调用路径中所有 LocalEnvironment 开始搜索，这样一来[①]，就可以引用到函数调用者的变量了。

这样的作用域称为**动态作用域**(dynamic scope)[②]。说实话，这点确实让人难以理解，所以在最近的语言中已经不流行这种方式了（Emacs Lisp 和 Perl 之类的 local 变量都是动态作用域）。

4.2.5 原生函数的形式参数

在调用的时候，crowbar 中描述的函数的形式参数是保存在局部变量中的，因此没有必要特别注意。原生函数的形式参数以数组的形式传递给原生函数，这样一来，即使 GC 在原生函数执行过程中启动，GC 也可以追溯到这些变量。

原生函数的实际参数存入 crowbar 栈中，传递给原生函数的只是头地址。

crowbar 的栈将占用更大的内存地址。因此，从前往后按顺序计算参数，即可将参数数组传递给原生函数。

① 即在函数被调用时。——译者注

② 关于动态作用域的策略，“对一个名字 x 的使用指向的是最近被调用但还没有终止且声明了 x 的过程中的这个声明”。（摘自 “龙书”[1]P19）——译者注

4.3 GC 的实现

之前介绍了标记 – 清除 GC 的基本原理和 crowbar 中对象引用的根。本节开始介绍在 crowbar 里运行着什么样的 GC 以及它的实现方式。

另外，与 GC 相关的代码大部分收录在 heap.c 中。

4.3.1 对象的管理方法

crowbar 中像字符串和数组这样的对象可以通过 CRB_Object 结构体保存在**堆**（heap）* 中。CRB_Object 需要逐个使用 MEM_malloc() 申请内存空间。

前面已经给出了 CRB_Object 结构体的定义，它在成员中持有指针 prev 和 next。由名字可以联想到，CRB_Object 是作为双向链表进行管理的。

CRB_Interpreter 中持有这个链表的头节点。为了方便管理堆相关的信息，我们定义了结构体 Heap。

```
typedef struct {
    int           current_heap_size;
    int           current_threshold;
    CRB_Object    *header;
} Heap;
```

header 指向 CRB_Object 链表的开头。current_heap_size 和 current_threshold 用于控制 GC 的启动时机。

4.3.2 GC 何时启动

在程序运行的过程中，标记 – 清除 GC 会在某个时机启动，释放不需要的对象。那么究竟要在何时启动 GC 呢？

其中一种方案就是内存不足的时候（即 malloc() 返回 NULL 时）。我们先不考虑 MEM_malloc() 在 malloc() 返回 NULL 时会调用 exit() 的情况，若是到了 malloc() 返回 NULL 的地步，那就说明内存空间是真的不足了，此时再运行 GC 为时已晚。像后面说到的那样，现在的标记 – 清除 GC 需要使用大量的栈空间，因此，如果真是到了 malloc() 返回 NULL 的地步，也不知道 GC 是否能

* C 语言使用 malloc() 申请内存空间，这里指可以以任意顺序申请或释放的内存区域。

启动。大多数的操作系统即使调用了 free() 函数，也不会将释放出来的内存空间还给操作系统，只是可以再次使用 malloc() 而已。因此即使 GC 将内存空间释放，其他应用（进程）也不能使用，所以 GC 要是坚持到内存被占满时才启动的话，会给其他程序带来很大的麻烦。

于是，crowbar 使用了这样一种方式，即耗费了一定量的内存后，就启动 GC。这个一定量在 Heap 结构体的 current_threshold* 中保存，初始值由宏[①]HEAP_THRESHOLD_SIZE 定义（#define），暂定为 256KB。

* threshold 是阈值的意思。

crowbar 每次创建对象，都要把所创建对象的大小值叠加到 CRB_Interpreter 中 Heap 结构体的 current_heap_size 上。这个大小值也就是要传给 MEM_malloc() 的大小值，所以这个值不包含 malloc() 或 MEM 模块的管理空间（毕竟是相似的）。

而且，在创建对象前，要先调用 check_gc()。

```
static void
check_gc(CRB_Interpreter *inter)

{
    /* 堆耗费量超过阈值的话…… */
    if(inter->heap.current_heap_size > inter->heap.current_threshold) {
        /* 启动 GC */
        crb_garbage_collect(inter);

        /* 设定下一个阈值 */
        inter->heap.current_threshold
            = inter->heap.current_heap_size + HEAP_THRESHOLD_SIZE;
    }
}
```

上面的函数中，如果堆的消耗量超过了当前的阈值就启动 GC。执行 GC 的时候，current_heap_size 的值会变小，将变小后的 current_heap_size 和 HEAP_THRESHOLD_SIZE 相加，就得出了下一个阈值。

至于函数 crb_garbage_collect() 就没有必要详细说明了。

```
void
crb_garbage_collect(CRB_Interpreter *inter)
{
    gc_mark_objects(inter);/* mark */
    gc_sweep_objects(inter);/* sweep */
}
```

① 预处理命令。——译者注

上面调用的 gc_mark_objects() 都做了哪些事，请参见代码清单 4-2。在清除了所有对象标记的基础上，从各个根（前面介绍过）开始调用 gc_mark()。

代码清单 4-2
gc_mark_objects()

```
static void
gc_mark_objects(CRB_Interpreter *inter)
{
    CRB_Object *obj;
    Variable *v;
    CRB_LocalEnvironment *lv;
    int i;

    /* 清除全部标记（mark）*/
    for (obj = inter->heap.header; obj; obj = obj->next) {
        gc_reset_mark(obj);
    }

    /* 全局变量 */
    for (v = inter->variable; v; v = v->next) {
        if (dkc_is_object_value(v->value.type)) {
            gc_mark(v->value.u.object);
        }
    }

    /* 局部变量 */
    for (lv = inter->top_environment; lv; lv = lv->next) {
        for (v = lv->variable; v; v = v->next) {
            if (dkc_is_object_value(v->value.type)) {
                gc_mark(v->value.u.object);
            }
        }

        gc_mark_ref_in_native_method(lv); /* ←这里稍后再做说明 */
    }

    /* crowbar 栈 */
    for (i = 0; i < inter->stack.stack_pointer; i++) {
        if (dkc_is_object_value(inter->stack.stack[i].type)) {
            gc_mark(inter->stack.stack[i].u.object);
        }
    }
}
```

gc_mark() 相关内容请参见代码清单 4-3。如果对象已经被标记，就直接 return（为了防止循环引用时出现死循环），然后将自己打上标记。如果是数组的话就遍历每个元素，并以此为参数递归调用 gc_mark()。

代码清单 4-3
gc_mark()

```
static void
gc_mark(CRB_Object *obj)
{
    if (obj->marked)
        return;

    obj->marked = CRB_TRUE;

    if (obj->type == ARRAY_OBJECT) {
        int i;
        for (i = 0; i < obj->u.array.size; i++) {
            if (dkc_is_object_value(obj->u.array.array[i].type)) {
                gc_mark(obj->u.array.array[i].u.object);
            }
        }
    }
}
```

4.3.3 清除阶段

在清除阶段释放那些链表中没有被标记的管理对象，并维护链表（见代码清单 4-4）。

代码清单 4-4
gc_sweep_objects()

```
static void
gc_sweep_objects(CRB_Interpreter *inter)
{
    CRB_Object *obj;
    CRB_Object *tmp;

    for (obj = inter->heap.header; obj; ) {
        if (!obj->marked) {
            if(obj->prev) {
                obj->prev->next = obj->next;
            } else {
                inter->heap.header = obj->next;
            }
            if (obj->next) {
                obj->next->prev = obj->prev;
            }
            tmp = obj->next;
            gc_dispose_object(inter, obj);
            obj = tmp;
        } else {
```

```
            obj = obj->next;
        }
    }
}
```

这其中调用的 gc_dispose_object() 把数组和字符串区分进行处理，释放了 CRB_Object 头地址指向的空间（见代码清单 4-5）。

代码清单 4-5
gc_dispose_object()

```
static void
gc_dispose_object(CRB_Interpreter *inter, CRB_Object *obj)
{
    switch (obj->type) {
    case ARRAY_OBJECT:
        inter->heap.current_heap_size
            -= sizeof(CRB_Value) * obj->u.array.alloc_size;
        MEM_free(obj->u.array.array);
        break;
    case STRING_OBJECT:
        if (!obj->u.string.is_literal) {
                inter->heap.current_heap_size -= strlen(obj->u.string.
string) + 1;
            MEM_free(obj->u.string.string);
        }
        break;
    case OBJECT_TYPE_COUNT_PLUS_1:
    default:
        DBG_assert(0, ("bad type..%d\n", obj->type));
    }
    inter->heap.current_heap_size -= sizeof(CRB_Object);
    MEM_free(obj);
}
```

补充知识　GC 现存的问题

crowbar 的 GC 最简单地实现了标记 – 清除 GC。因为是最简单的实现，所以还存在以下问题。

1. 运行 GC 时，程序会停止运行。
2. 标记（mark）时的递归调用会消耗大量的栈空间。

首先是问题 1。因为是简单实现的标记 – 清除算法，所以在进行标记 – 清除时会完全停止主程序的运行（crowbar 也是如此）。为了避免（减轻）这个问题，大概有以下两种方法。

- **让 GC 和主程序异步（并行）执行**
- **使用分代式 GC 技术**

如果让 GC 和主程序异步（并行）执行，虽然 GC 占用 CPU 的总耗时不变，但是可以避免程序停止的情况，这对于一个互动程序来说是非常重要的。但是，实际上 GC 在异步执行时，肯定会发生当 GC 标记对象时，主程序中的对象引用关系同时发生改变的情况，这样一来有些对象就可能被遗漏，没有打上标记。避免这种情况的方法是有的（请用“write barrier”等词搜索一下），但是实现起来有点难度。

分代式的 GC 技术基于“经过一段时间后依然存活的对象有可能一直存活下去”的经验，将对象分为不同世代进行管理。经过一段时间后依然被引用的对象，被当做“老年代”处理，对于老年代的对象，GC 执行得不会很频繁。

比如需要编辑一个很大的文本时，实际需要编辑的内容只是全文的一小部分，但是在简单实现的 GC 中，却要对全部文本进行标记，这样会做很多无用功。而分代式的 GC 避免了这样的无用功，缩短了对于 GC 来说很重要的时间，提高了总体的处理速度。

问题是，从新生代的对象向老年代对象转变的过程中，在取消新生代对象的标记，又没有标记为老年代时，就发生了漏标记的情况。这当然是不允许的，必须要有对策。

下面我们再来看一下标记－清除的第二个问题，即进行标记（mark）时的递归调用会消耗大量的栈空间。明明是因为内存不足才启动的 GC，但是 GC 又消耗了大量的栈空间，这让人有种本末倒置的感觉。数据结构决定了到底要消耗多少栈空间，像“将巨大的文件全部读入链表中，再进行一些处理”的程序，可想而知是很简单的（特别是使用脚本语言来做）。大量的小对象链接构成链表，再递归进行标记，光是对象的数量就足以占用大量的栈空间了。

我们使用称为链接反转法的思考方式来避免这个问题。在标记对象时，为了方便递归，以及在一个对象标记结束后能够更容易返回到持有它的对象，我们在栈中使用局部变量来记录已经处理了对象中的哪个引用。链接反转法用下面的方法实现在对象内的记录。

- **按照对象 A →对象 B 的顺序地标记，在移动到下一个对象 C 时，将对象 B 中指向 C 的引用指向 A。**
- **每个对象都要增加成员“已经处理了哪个引用”。**

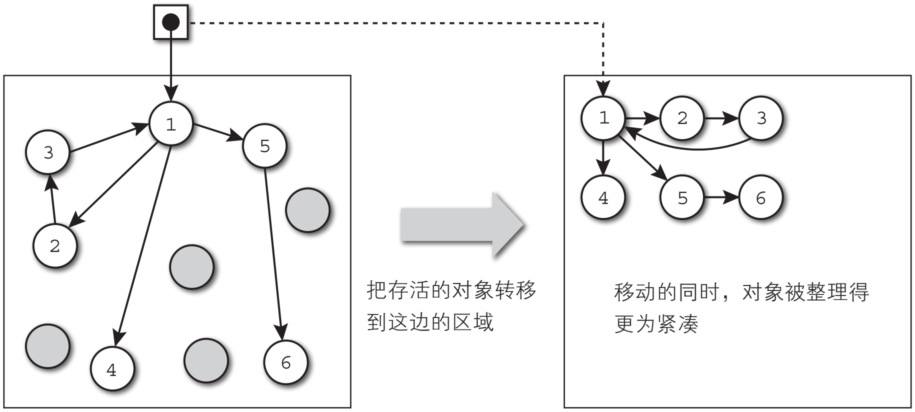
补充知识　Copying GC

Copying GC 是一种已知的（经典的）垃圾回收器实现方法。 Copying GC 有以下策略（如图 4-10）。

- **一开始就创建一个大的堆区域，并将它一分为二。**
- **创建对象的时候，从其中一半的区域划分出内存投入使用。**
- **当另一半区域被装满时，使用和标记－清除的标注阶段相同的方式跟踪对象，只将生存的对象复制到另一半区域中即可。此时，由于对象的地址发生了变化，因此需要维护指向它们的所有指针。**

- **复制之后，将对象复制的目标区域切换为创建对象时使用的区域。当这半边区域被装满时，再向另一半区域复制，如此循环往复。**

图 4-10
Copying GC



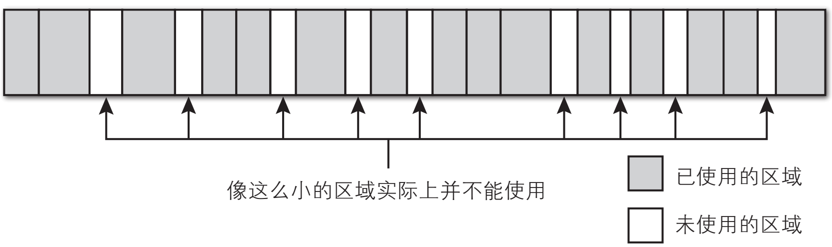
这个方法一目了然，它把最初的内存区域一分为二，在一个时间点只使用其中的一半，另外一半内存就浪费了。

看上去好像是一个效率极差的方法，但是这个方法和标记 – 清除的 GC 相比有以下优点。

- **复制对象的同时进行压缩（compaction），由此，消除了碎片（fragmentation），提高了虚拟内存和高速缓存的效率。**
- **舍弃了标记 – 清除的清除过程，因此，在生存对象占比较小的情况下效率较高。**

碎片是指像 `malloc()` 这样的函数多次对内存进行申请 / 释放的时候，内存如图 4-11 所示，出现极小且不连续的空间的状态。

图 4-11
碎片



这样一来，对象之间存在间隙的内存区域实际上是不能使用的，这就造成了内存的浪费 *。

* 去除这些没用的间隙，把对象存储空间变紧凑的操作叫作精简（compaction）。

Copying GC 在复制的时候将这些间隙消除。另外，一边追溯指针一边复制的方式使得相互指向的对象很可能被复制到临近的区域。因此，同时使用的对象也很有可能被放在一起，并写到虚拟内存的同一页中以减少翻页的几率，提升了性能。

4.4 其他修改

接下来要介绍的是 crowbar ver.0.2 中除了 GC 之外的其他需要修改的地方。

4.4.1 修改语法

对语法作了如下修改。

- 数组——表达式之后可以加 [表达式]。
- 方法调用——表达式后面可以加方法名（参数列表）。
- 自增 / 自减——表达式后面可以加 ++ 或 --。

增加了很多可以追加在表达式后面的语法。在语法结构上引入了非终结符 postfix_expression（这个名字是从 K&R[2] 的附录 C 中得来的）。

```
unary_expression
    : postfix_expression
    | SUB unary_expression
    ;
postfix_expression
    : primary_expression
    /* 引用数组元素。LB 和 RB 是“[”和“]”*/
    | postfix_expression LB expression RB
    /* 调用方法 */
    | postfix_expression DOT IDENTIFIER LP argument_list RP
    | postfix_expression DOT IDENTIFIER LP RP
    /* 自增，自减 */
    | postfix_expression INCREMENT
    | postfix_expression DECREMENT
    ;
```

句柄 LB 和 RB 是 Left Bracket 和 Right Bracket 的简写，分别代表“[”和“]”。

根据这个语法结构创建的结构体如下所示（crowbar.h）。因为都是表达式，所以都加入到了 Expression 结构体的联合体中。

```
/* 引用数组元素 */
typedef struct {
    Expression *array;
    Expression *index;
} IndexExpression;

/* 自增 / 自减 */
typedef struct {
    Expression *operand;
} IncrementOrDecrement;

/* 调用方法 */
typedef struct {
    Expression      *expression;
    char            *identifier;
    ArgumentList    *argument;
} MethodCallExpression;
```

4.4.2 方法的模拟

crowbar ver.0.2 的数组配备了下面这些“像方法一样的函数”。

```
# 给数组增加元素
a.add(3);

# 取得数组的大小
size = a.size();

# 改变数组的大小
a.resize(10);
```

另外，（顺便）也给字符串添加了方法。

```
# 取得字符串的长度
len = "abc".length();
```

之所以说起方法的“模拟”，是因为现在在 crowbar 中还没有为类型（或者对象）分配方法的通用手段。book_ver.0.2 的实现虽说属于偷工减料，但也算是用了嵌入代码的方式。（截取自 eval.c，见代码请单 4-6。）

代码清单 4-6
eval_method_call_expression()

```
static void
eval_method_call_expression(CRB_Interpreter *inter, CRB_LocalEnvironment *env,
                            Expression *expr)
{
    CRB_Value *left;
    CRB_Value result;
    CRB_Boolean error_flag = CRB_FALSE;
    eval_expression(inter, env, expr->u.method_call_expression.expression);
    left = peek_stack(inter, 0);

    if (left->type == CRB_ARRAY_VALUE) {
        if (!strcmp(expr->u.method_call_expression.identifier, "add")) {
            CRB_Value *add;
            check_method_argument_count(expr->line_number,
                                        expr->u.method_call_expression
                                        .argument, 1);
            eval_expression(inter, env,
                            expr->u.method_call_expression.argument
                            ->expression);
            add = peek_stack(inter, 0);
            crb_array_add(inter, left->u.object, *add);
            pop_value(inter);
            result.type = CRB_NULL_VALUE;
        } else if (!strcmp(expr->u.method_call_expression.identifier,
                           "size")) {
            check_method_argument_count(expr->line_number,
                                        expr->u.method_call_expression
                                        .argument, 0);
            result.type = CRB_INT_VALUE;
            result.u.int_value = left->u.object->u.array.size;
        } else if (!strcmp(expr->u.method_call_expression.identifier,
                           "resize")) {
            CRB_Value new_size;
            check_method_argument_count(expr->line_number,
                                        expr->u.method_call_expression
                                        .argument, 1);
            eval_expression(inter, env,
                            expr->u.method_call_expression.argument
                            ->expression);
            new_size = pop_value(inter);
            if (new_size.type != CRB_INT_VALUE) {
                crb_runtime_error(expr->line_number,
                                  ARRAY_RESIZE_ARGUMENT_ERR,
                                  MESSAGE_ARGUMENT_END);
            }
            crb_array_resize(inter, left->u.object, new_size.u.int_value);
            result.type = CRB_NULL_VALUE;
```

```
            } else {
                error_flag = CRB_TRUE;
            }

        } else if (left->type == CRB_STRING_VALUE) {
            if (!strcmp(expr->u.method_call_expression.identifier, "length")) {
                check_method_argument_count(expr->line_number,
                                            expr->u.method_call_expression
                                            .argument, 0);
                result.type = CRB_INT_VALUE;
                result.u.int_value = strlen(left->u.object->u.string.string);
            } else {
                error_flag = CRB_TRUE;
            }
        } else {
            error_flag = CRB_TRUE;
        }
        if(error_flag) {
            crb_runtime_error(expr->line_number, NO_SUCH_METHOD_ERR,
                              STRING_MESSAGE_ARGUMENT, "method_name",
                              expr->u,method_call_expression.identifier,
                              MESSAGE_ARGUMENT_END);
        }
        pop_value(inter);
        push_value(inter, &result);
    }
```

说点题外话，最初在获取数组元素数的时候使用的是 get_size()，作为一个 getter 方法，为了统一性着想，取名时应为 get_xxx() 的形式。但是，考虑到这个方法使用频繁，并且经常要放在 for 语句中使用，方法的名字如果太长使用起来不太方便，因此还是取名为 size() 了。

我常常在想，虽然统一性很重要，但是方便性一样重要。不论是制作语言还是程序库，要想兼顾这两个方面还真是不容易。

不过，取得数组大小为什么不用方法，而只用了 length ？取字符串长度时用的是 length()，而 Vector 和 ArrayList 这样取大小的为什么用的是 size() 呢？我想这只是为了与之前的内容兼顾而产生的不统一。

4.4.3 左值的处理

在一般的表达式中，一个变量表示该变量储存的值。比如，将 5 赋值给 a，

当表达式中的 a 替换为 5 时表达式的结果保持不变。但是，变量在赋值的左边时，此时将

```
a = 10;
```

变为

```
5 = 10;
```

是行不通的。总之，变量在赋值语句的左边时，变量代表的不是它储存的值，而是储存值的地方（即变量的内存地址），我们称其为**左值**（left value）。

book_ver.0.1 中，赋值语句的左边只能放变量。赋值语句的语法规则如下所示。

```
expression
        : IDENTIFIER ASSIGN expression
        ;
```

但是在引入了数组之后，赋值语句的左边就可以写稍微复杂一点的表达式了。

```
# 给二维数组 a 赋值，其中一个下标为以 b[i] 为参数
# 调用函数 func() 后的返回值
a[i][func(b[i])] = 5;
```

当然，使用老的语法结构不能对应这种情况，于是，我们将语法结构改写成下面这样。

```
expression
        : postfix_expression ASSIGN expression
        ;
```

左边变成了 postfix_expression，而现在能成为赋值语句对象的，只有 primary_expression（变量名）和 postfix_expression（数组元素）了。

接着，在 eval.c 中首先计算右边的值，然后调用 get_lvalue() 函数取得左边表达式的地址，详见代码清单 4-7。并且，这里调用的 peek_stack() 函数会在不清除栈的情况下获取值，这样一来右边的值会作为赋值表达式整体的值残留在栈中。

代码清单 4-7 eval_assign_expression

```
static void
eval_assign_expression(CRB_Interpreter *inter, CRB_LocalEnvironment *env,
                       Expression *left, Expression *expression)
```

```
{
    CRB_Value *src;
    CRB_Value *dest;

    /* 首先计算右边值 */
    eval_expression(inter, env, expression);
    src = peek_stack(inter, 0);

    /* 取得左边的地址 */
    dest = get_lvalue(inter, env, left);
    *dest = *src;
}
```

那么，eval_assign_expression() 中调用的函数 get_lvalue() 又是什么样子呢？请见代码清单 4-8。将标识符和数组分开处理，如果是数组的话，可以利用 get_array_element_lvalue()（见代码清单 4-9）函数返回数组元素对应的地址。

代码清单 4-8
get_lvalue()

```
CRB_Value *
get_lvalue(CRB_Interpreter *inter, CRB_LocalEnvironment *env,
           Expression *expr)
{
    CRB_Value *dest;

    if (expr->type == IDENTIFIER_EXPRESSION) {
        dest = get_identifier_lvalue(inter, env, expr->u.identifier);
    } else if (expr->type == INDEX_EXPRESSION) {
        dest = get_array_element_lvalue(inter, env, expr);
    } else {
        crb_runtime_error(expr->line_number, NOT_LVALUE_ERR,
                          MESSAGE_ARGUMENT_END);
    }
    return dest;
}
```

代码清单 4-9
get_array_element_lvalue()

```
{
    CRB_Value array;
    CRB_Value index;

    /* 运算 [] 左边的表达式 */
    eval_expression(inter, env, expr->u.index_expression.array);
    /* 运算 [] 中的表达式 */
    eval_expression(inter, env, expr->u.index_expression.index);
    /* 取得两个变量的值 */
    index = pop_value(inter);
    array = pop_value(inter);
```

```
    /* 检查数据类型 */
    if (array.type != CRB_ARRAY_VALUE) {
        crb_runtime_error(expr->line_number, INDEX_OPERAND_NOT_ARRAY_ERR,
                          MESSAGE_ARGUMENT_END);
    }
    if (index.type != CRB_INT_VALUE) {
        crb_runtime_error(expr->line_number, INDEX_OPERAND_NOT_INT_ERR,
                          MESSAGE_ARGUMENT_END);
    }

    /* 检查下标范围 */
    if (index.u.int_value < 0
        || index.u.int_value >= array.u.objext->u.array.size) {
        crb_runtime_error(expr->line_number, ARRAY_INDEX_OUT_OF_BOUNDS_ERR,
                          INT_MESSAGE_ARGUMENT,
                          "size", array.u.object->u.array.size,
                          INT_MESSAGE_ARGUMENT, “index”, index.u.int_value,
                          MESSAGE_ARGUMENT_END);
    }
    /* 返回地址 */
    return &array.u.object->u.array.array[index.u.int_value];
}
```

另外，在没有左值的情况下，取得数组元素值的引用时调用的函数 eval_index_expression()，它的内部也使用了 get_array_element_lvalue()。

```
static void
eval_index_expression(CRB_Interpreter *inter,
                      CRB_LocalEnvironment *env, Expression *expr)
{
    CRB_Value *left;

    left = get_array_element_lvalue(inter, env, expr);

    push_value(inter, left);
}
```

自增 / 自减运算符也同样适用于使用 get_lvalue() 获取目标变量的地址。

4.4.4 创建数组和原生函数的书写方法

前面已经说过，用字面量可以创建数组 {1, 2, 3}，用原生函数 new_array() 也可以。该原生函数的定义请见代码清单 4-10。

代码清单 4-10
new_array()

```
/* 递归调用子例程 */
CRB_Value
new_array_sub(CRB_Interpreter *inter, CRB_LocalEnvironment *env,
              int arg_count, CRB_Value *args, int arg_idx)
{
    CRB_Value ret;
    int size;
    int i;

    if(args[arg_idx].type != CRB_INT_VALUE) {
        crg_runtime_error(0, NEW_ARRAY_ARGUMENT_TYPE_ERR,
                          MESSAGE_ARGUMENT_END);
    }
    size = args[arg_idx].u.int_value;

    ret.type = CRB_ARRAY_VALUE;
    ret.u.object = CRB_create_array(inter, env, size);

    if (arg_idx == arg_count-1) {
        for (i = 0; i < size; i++) {
            ret.u.object->u.array.array[i].type = CRB_NULL_VALUE;
        }
    } else {
        for (i = 0; i < size; i++) {
            ret.u.object->u.array.array[i]
                = new_array_sub(inter, env, arg_count, args, arg_idx+1);
        }
    }

    return ret;
}

/* 原生函数本体 */
CRB_Value
crb_nv_new_array_proc(CRB_Interpreter *interpreter,
                      CRB_LocalEnvironment *env,
                      int arg_count, CRB_Value *args)
{
    CRB_Value value;

    if (arg_count < 1) {
        crb_runtime_error(0, ARGUMENT_TOO_FEW_ERR,
                          MESSAGE_ARGUMENT_END);
    }

    value = new_array_sub(interpreter, env, arg_count, args, 0);
```

```
        return value;
    }
```

首先，这次的修改在原生函数的参数中增加了 CRB_LocalEnvironment。

上面代码中的处理只是递归地创建了数组的数组，数组所需内存空间的开辟工作由 CRB_create_array() 完成。同时还为原生函数增加了形式参数，从而使 CRB_LocalEnvironment 可以作为参数传递进去。那么，CRB_LocalEnvironment 的用途是什么呢？在创建多维数组的时候，会多次调用 new_array_sub() 开辟内存空间。如果 GC 在进行上述操作的中途启动的话，就可能立刻释放掉刚分配好的数组。因此，为了把在函数中创建的对象标记为不回收，我们需要把 CRB_LocalEnvironment 传递到函数中。

具体来说，CRB_LocalEnvironment 的成员 ref_in_native_method 中保存了在原生函数中创建的对象（具体请参考 4.2.4 节的定义）。

类型 RefInNativeFunc 的定义如下，它以链表的形式保存着指向对象的数组。

```
typedef struct RefInNativeFunc_tag {
    CRB_Object *object;
    struct RefInNativeFunc_tag *next;
} RefInNativeFunc;
```

这种设计的问题在于，创建于原生函数内部的对象在函数结束前不能被释放。如果在原生函数中存在大量多次循环时，就会出现很多对象在创建后立即被丢弃的现象。有些对象在函数结束前不能被释放，造成了内存空间的浪费。但是，如果只是在原生函数内部使用的对象，那么比起使用麻烦的 crowbar 对象，使用 C 的 malloc() 更方便。因此，实际情况下像上面那样的问题并不多见，所以还是维持了这样的设计。

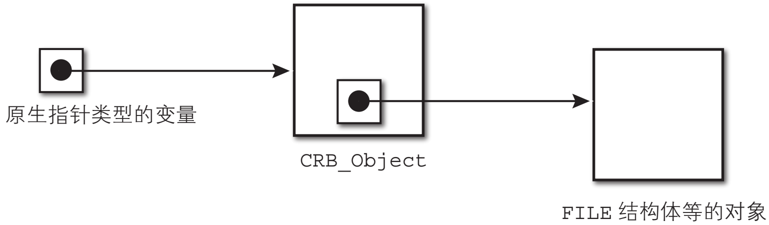
4.4.5　修改原生指针类型

crowbar book_ver.0.1 的原生指针类型的值（CRB_Value 结构体）中包含了 void* 的指针和数据类型的标识信息。在 3.3.9 节中提到过，这个方法在“一个地方用 fclose() 关闭了文件指针，可能还会在另一个地方被使用”的情况下

会存在问题。

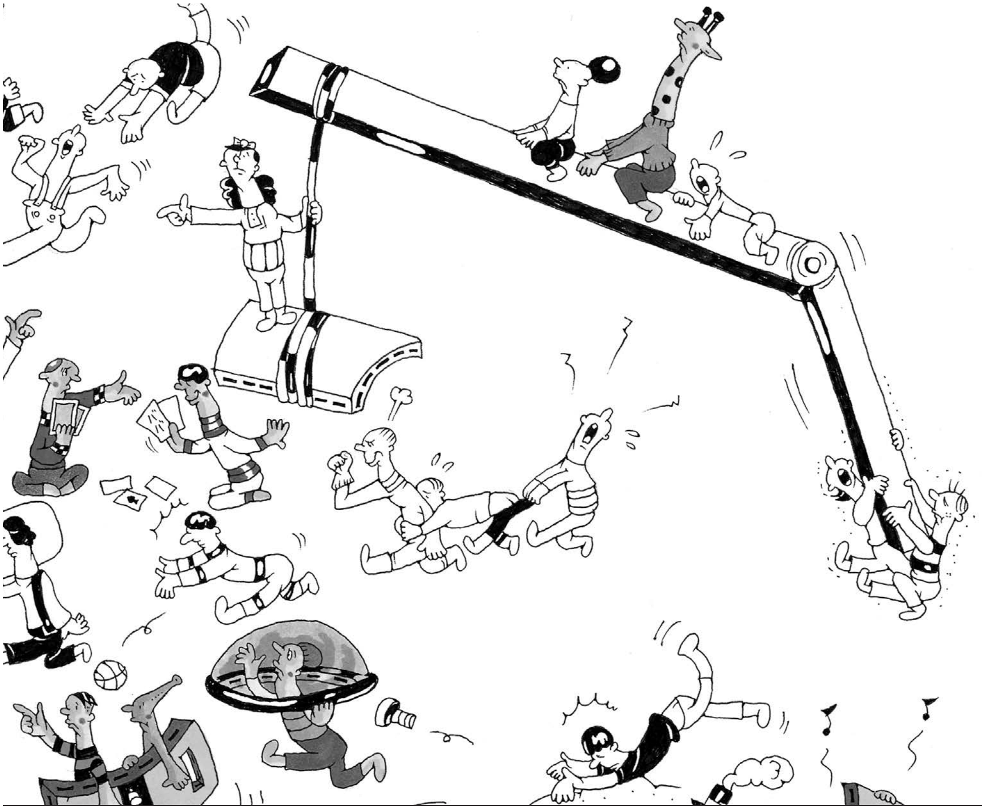
在 book_ver.0.2 中，原生指针类型的值不再指向实际的原生对象（FILE 结构体等），而是在中间加入了一个 crowbar 对象进行“隔离”（如图 4-12）。

图 4-12
原生指针的构造（修改版）



运用上面的方法，例如在调用 `fclose()` 的时候，将 `CRB_Object` 中指向原生对象的指针同时置为 `NULL`，就能够立刻判断出文件已被关闭。也许有人会想，在以前的做法中，没有将 `CRB_Value` 直接指向 `FILE` 结构体，而是在它们中间夹了一个别的结构体，不是也起到同样的作用了吗？假设真的这么做了，那么在对这个结构体进行 `free()` 的时机就又成了一个问题。何不借着实现 GC 的机会，让它们中间夹一个可以作为 GC 目标的对象呢？

顺便说一下，虽然这样一来就可以实现针对原生指针类型的**终结器**（finalizer）了（GC 在释放原生指针类型的对象时调用之前注册的函数就可以了），但是由于标记 – 清除型的 GC 中终结器“不知何时启动”，因此使用 crowbar 的用户最好不要编写依赖于终结器的程序。但是对于一门编程语言来说，提供终结器也不是什么坏事。



第 5 章

中文支持和 Unicode

5.1 中文支持策略和基础知识

在第 4 章中提到的 crowbar 是不支持中文的。一个标榜着像 Perl 一样的语言，怎么可以不能正确处理含有本地语的中文文本文件呢？本章开始，我们就来看一看汉化的处理方式。

5.1.1 现存问题

看了“crowbar 是不支持中文的”这句话后，肯定有人会问：“这是真的吗？”举个例子吧。

```
print(“你好 \n”);
```

这行代码在大多数环境下应该可以正常运行。但 crowbar 处理器其实并没有意识到字符编码的问题，只不过是直接输出了一个含有字符串字面量的字节序列，根本不能说它支持中文。具体有以下这些问题。

1. GB2312 环境下的 0x5C 问题

现在的 crowbar 代码是采用 GB2312 编码保存的，因此可能会因为特殊字符而引起误动作。具体来说就像中文的“沺”和“晒”。

比如，“晒”的 GB2312 编码是0x955C。第二个字节的 5C，是反斜杠的编码，这会导致字符串字面量“晒”被编码成“"”和“\”组成的字符串。这里只是在说字符串字面量的话题，crowbar 程序在读取来自外部文件和标准输入的字符串时不会受到影响。

2. length() 方法

我们在 crowbar book_ver.0.2 中为字符串引入了 `length()` 方法。现在的这个函数在执行代码 `"北京欢迎您".length()` 时会返回 10。这种设计与 C 的 `strlen()` 相同，但这样的设计在 crowbar 中几乎派不上用场。“北京欢迎您”是 5 个字，调用 `length()` 函数就应该返回 5 才对（在 C 语言中，很多情况下要有很强的“字节数”意识，因此 `strlen()` 的设计不得不说还是挺方便的）。

现在的 crowbar 中，字符串只有 `length()` 方法，今后还要引入截取字符串的 `substr()` 等方法，在那个时候这个问题就更为重要了。

既然现在 crowbar 的实现有以上的问题，那怎么解决好呢？想要考虑清楚这个问题，需要很多基础知识。下面我们就对以下几项进行简单说明。

5.1.2 宽字符（双字节）串和多字节字符串

宽字符（wide character）和**多字节字符**（multibyte character）的说法源于 C 语言的用语。

多数人在编写 C 语言程序的时候使用 `char` 数组来表示字符串。可是，`char` 只能存储 1 个字节（通常是 8 位），存不下一个中文的字符。因此，中文字符的存储都是使用 GB2312（EUC，即扩展 UNIX 编码，Expanded UNIX code）*、GBK、UTF-8 等**编码**。这种用多个字节保存一个字符的字符串形式称为**多字节字符串**。

* EUC-CN 是 GB2312 最常用的表示方法。浏览器编码表上的 GB2312，通常都是指 EUC-CN 表示法。

但是，这种保存方式存在一个问题，即不从 `char` 数组的开头开始看，就找不到哪里是字符的分割点。C 语言使用 `str[i]` 获取字符串中一个字符时，也不知道要取的是一个英文数字字符还是中文字符的第一个字节或第二个字节。在这种情况下，如果要制作一个编辑器，在按退格（backspace）键的时候，如果只是删除了中文字符的第二个字节，那么后面跟着的内容就全都变成乱码了。实际上，以前的很多编辑器都有这个问题。

GB2312 的 `0x5C` 问题大概也是这个原因。如果只看一个中文字符的第二个字节，就会被误认为是“\”。

只保存 8 位的 `char` 类型显然不能满足需求，要是有一个内存空间足够的类型用来保存中文等字符就好了。C 语言中的 `wchar_t` 类型，就是一个有足够内存空间的类型（或者说是我们期望的类型）。

以 `wchar_t` 数组表示一个字符串的方式叫作**宽字符串**。正如我们期待的，`str[i]` 可以取出这个字符串中第 i+1 个字符 *。

* 为什么不是取出第 i 个字符呢？因为 C 的数组下标从 0 开始。

在 C 语言的标准中，并没有规定 `wchar_t` 到底是几个字节，以及 `wchar_t` 在存储文字时用什么编码格式。在基于 UNIX 的 gcc 中执行 `sizeof(wchar_t)` 的话返回 4，在 Windows（MinGW 的 gcc 或 VC++ 等）中返回 2。另外，在 C 代码中如果想要定义宽字符和宽字符串的话，宽字符使用 `L'a'`，宽字符串使用 `L"abc"`。

宽字符串 L"abc" 在 wchar_t 是 4 字节的环境中,需要消耗 16 字节的内存空间（最后的 Null 字符也是 4 个字节）。我们暂且不说这种存储方式进行类型转换的时候会发生编译错误，存储在内存上的 L"abc"，以字节为单位去看的话中间可能会出现很多 0，会被误判为字符串末尾的 Null 字符。因此，宽字符串不能使用 strcpy() 和 strcmp() 之类的函数，必须要用 wcscpy() 代替 strcpy()，用 wcscmp() 代替 strcmp()。

补充知识　**wchar_t 肯定能表示 1 个字符吗?**

我们在 5.1.2 节中说到，wchar_t 类型有足够的内存空间保存中文等字符（或者说是我们想要的类型）。可能会有人不满意这种模棱两可的说法，那么就让我们来确认一下 C 语言标准中的定义吧。

在 IOS C99（ISO/9899:1999）中关于 wchar_t 的最小内存空间记载如下（7.18.3）:

> **wchar_t（参考 7.17 节）用带符号整数类型定义的情况下，WCHAR_MIN 的值不得小于 –127，WCHAR_MAX 的值不得大于 127。**
> **wchar_t 用无符号整数类型定义的情况下，WCHAR_MIN 的值必须是 0，WCHAR_MAX 的值不得大于 255。**

WCHAR_MIN 和 WCHAR_MAX，顾名思义是 wchar_t 最大值和最小值的常量。总之，在标准中保证了 wchar_t 的大小只有 1 个字节，但是从 wchar_t 的定义本身来看，规定如下:

> **wchar_t 值的范围要能够容纳处理器所支持的区域设置中最大的扩展字符集（包含全部编码要素的整数类型）。**

至少在我看来这句话的意思是，wchar_t 的大小必须能存下所支持字符集的任何一个字符。

但是，实际上 Windows 的 wchar_t 只有两个字节，所以不能表示超过 UCS2 范围的字符。因此，至少在 Windows 中，wchar_t 类型不能表示一个字符（实际上 Windows 的宽字符串是 UTF-16）。可见世上没有最理想的事。

5.1.3　多字节字符 / 宽字符之间的转换函数群

字符的表示方法有多字节字符和宽字符两种，具体内容在前面的小节已经介绍过了，相信大家也应该清楚了它们的使用方法。

宽字符串把字符保存在 wchar_t 中，因此前面说到的制作编辑器、为字符串

添加 length() 和 substr() 方法都不成问题。可是，像英文数字这种平时只需要 1 个字节表示的字符，在这里也需要占用 sizeof(wchar_t) 大小的内存空间了。

因此，在保存文件的时候，不推荐使用宽字符形式。宽字符是 C 语言的概念，显而易见文件和编程语言是独立的。之前，"用同样的字节数来表示所有的字符"这种想法，在编程时处理字符串是很方便的，但是作为文件处理方式的时候就不再占据优势了，反而只会单纯地占用容量 *。于是，crowbar 的处理器有必要进行如下的变化。

* 在存储容量的价格方面，考虑到内存要比硬盘贵，节约内存空间也是理所当然的。

- 从文件和标准输入中输入字符串时，从多字节字符转换为宽字符。
- 向文件和标准输出中输出字符串时，从宽字符转换为多字节字符。

这样一来，现在的状态就成了"在 crowbar 外部使用多字节字符，内部使用宽字符"。在 C 语言中可以使用以下的函数群进行转换操作。这些函数是以 ISO C95 标准进行了标准化的函数群，这些函数不仅名字具有标准性，十分好记，详细的设计也编成了手册可在线阅读。在这里，我只对它们的基本功能进行说明。

■ **多字节字符向宽字符转换**

- `int mbtowc(wchar_t *pwc, const char *s, size_t n);`

  从多字节字符的字节序列中读取出代表一个字符的字节（最大 n 字节），将转换后的宽字符的指针保存在变量 pwc 中。功能和名字一样，mb（多字节字符）转换为 wc（宽字符）。

- `size_t mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps)`

  在 C 语言的规格书中将多字节字符定义（5.2.1.2）为"可以携带依赖于转换状态的表现形式"。

  利用**换码序列**（escape sequence）这种特殊的字节序列进行编码间的切换，当文字中出现了换码序列时，它之后的内容就是中文，而在中文中换码序列之后的内容就是 ASCII 字符。

  用这种方式将某个多字节字符转换为宽字符时，必须要知道它现在是什么状态。

  如果使用 mbtowc() 从开头对多字节字符串进行处理的话，那么 mbtowc() 会在内部记住当前的状态，并且根据状态转换出适当的宽字符。可

能很多人会觉得这样就已经很好了。但是，如果这段使用了 mbtowc() 的转换程序用来进行一个多线程的字符串处理时，mbtowc() 的这种“记状态”的机制就被破坏了。

于是 mbrtowc() 将保存当前状态的内存空间交给了调用者。这个空间的类型是 mbstate_t。在最初调用的时候，mbstate_t 可以使用 memset() 等函数进行清零。

函数的名字是 mbrtowc()，中间加了一个 r。我想这个 r 可能是 reentrant 的 r。

- `size_t mbstowcs(wchar_t *dest, const char *src, size_t len)`

  上述两个函数是用来转换文字的，这个函数可以将整个字符串一起处理。于是函数名字在 mb 和 wc 后面都加上了 s。

  函数将多字节字符串 src 转换为宽字符串，字符最大为 n 字节，结果保存在 dest 指针中。函数的返回值会返回写入到 dest 的宽字符个数（不包含结尾的 L'\0'），如果只想知道宽字符的字符个数的话，经常使用的方法是给 dest 传 NULL 并获取返回值。

- `size_t mbsrtowcs(wchar_t *dest, const char **src, size_t len, mbstate_t *ps)`

  和转换单个字符的函数一样，加上 r 就变成了 reentrant 版。src 变成了指针的指针，是因为要在转换过程中将 src 移动到下一个要转换的多字节的开头[①]。

■ 宽字符向多字节字符转换

- `int wctomb(char *s, wchar_t wc)`

  可见，这是反过来将 wc 转换为 mb 的函数，也就是将一个宽字符转换为多字节字符的字节序列的函数。

- `size_t wcrtomb(char *s, wchar_t wc, mbstate_t *ps)`

  加上 r 就是 reentrant 版。

- `size_t wcstomb(char *dest, const wchar_t *src, size_t n)`

  加上 s 就是字符串版。

① 这样做是为了对应多线程的情况。——译者注

- `size_t wcsrtombs(char *dest, const wchar_t **src, size_t len, mbstate_t *ps)`

  s 和 r 都加上，变成了字符串版的 reentrant 版。

为了能够使用这些函数，在 Windows（MinGW）中编译时必须配置启动项 -lmsvcp60。

5.2 Unicode

前面章节已经做了简要说明，现在的 Linux 和 Windows 中，宽字符大多使用的是 Unicode。作为 crowbar 处理器，如果想要制作一个转换宽字符和多字节字符的程序库，可以不用考虑实际的 wchar_t 中存储的字符到底是什么字符编码。但是，作为一名程序员是不能回避 Unicode 问题的，所以本章将对此进行介绍。

5.2.1 Unicode 的历史

Unicode 是由 Xerox 提出，并由 Unicode Consortium 制定的字符编码。

Unicode 最初的内容是“用 16 位表示全世界所有的字符”，所以，中国、日本、韩国使用的汉字只要字形是一样的，都会分配到同样的字符编码（Unicode 中正确的叫法应该是码位，即 code point）。截止到 1990 年 12 月的最终草案，汉字分配了 0x4000~0xE7FF，共 18739 个字符。

在中国，GB2312（EUC-CN）能表示的汉字总共有 6763 字，由此得知，在 Unicode 出现之前使用普通 PC 可以处理的汉字，全部可以用 Unicode 表示。但是，考虑到经常有一些人名或者古汉语中的字符无法用国标汉字表示，因此即使是 18739 个字也不一定够用，更何况这个范围内还包含了日本和韩国使用的汉字[①]。

另外，对于日本人来说，假名等分配了 0x3000~0x3FFF，共 4096 个字符。假名的字符很少，这个范围已经足够了，但是韩语一个字符的形状由初声、中声、终声的排列组合决定，初声 19 种，中声 21 种，终声 27 种，加上有些字符

① 有时候，三个国家使用的汉字即使字形相同也是不一样的，因此分配了不同的码位。——译者注

没有终声的情况，一共有 19 × 21 × (27 + 1) = 11172 个字符 [4]。这么多字符，Unicode 当然适应不了了 *。

* 1987 年的时候，韩国国内的标准 KSC 5601 修订版中韩语只有 2350 个字符，同一时期的 Unicode 已经包含了这些字符，如果是日常使用的话，16 位的 Unicode 也没什么问题。

如此一来，结果就是在现有的 16 位 Unicode 上进行扩充（现在是 21 位），以 16 位为范围收录的一套字符作为 UCS2（表示 Universal Coded-Character Set 的 2 位版）进行标准化，收录不下的所有字符以 4 字节表示，这就是 UCS4。

5.2.2 Unicode 的编码方式

如前所述，Unicode 本来想用 16 位来表示世界上所有的字符（但实际上收录不下）。那么，在内存和磁盘中保存字符串的时候，只能 2 字节一组表示 1 个字符吗？也不一定。

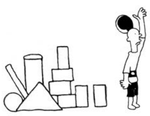
我们首先介绍一下**字符集**（character set）和**编码方式**（正确的说法应该是字符符号化方式）。

计算机想要处理字符，首先要决定什么样的字符是要处理的对象，其次就是为这些字符分配编号，这就是字符集。为每个字符分配的编号在 Unicode 中称为**码位**（code point）。例如中文“啊”的码位是 0x554A，记作 U+554A。

但是，这些字符想要在内存或者磁盘上表示就是另外一码事了。编码方式是指规定以何种方式将逻辑上的码位值以字节或位的方式表示出来。虽然很可能某两种编码方式的目标字符集相同，但因为编码方式和字符在字符集中的顺序不同，因此在内存上的表现形式也不同。

Unicode 的编码方式首先要考虑的是，Unicode 是 16 位的，要为一个字符分配 2 字节，这种方法被称为**UTF-16**。

但是，假如使用 C 语言中 2 字节的整数类型（short 等）表示 1 个字符，内存上的表示方式会根据环境的字节序产生变化。因此，UTF-16 根据字节序的不同分为 UTF-16BE（大尾序）和 UTF-16LE（小尾序）两种。中文“啊”，用 UTF-16BE 表示为 `0xB0 0xA1`，用 UTF-16LE 表示为 `0xA1 0xB0`。另外，在和其他 PC 通信的时候，如果不知道字节序也很麻烦。可以在 UTF-16 的字符串开头加上字节序的标识（这种标识叫作**BOM**，即 Byte Order Mark，值为 U+FEFF）。读取带 BOM 的 UTF-16 字符串时，如果第一个字符是 0xFE 0xFF 就是 UTF-16BE，如果是 0xFF 0xFE 就是 UTF-16LE。



但是，UTF-16 如果要表示字母 A（码位为 `U+0041`）也要消耗 2 个字节，这样在英语圈的人（只使用 ASCII 的人）看来，字符串在内存和磁盘上的消耗突然变成了原来的 2 倍。而且，字符串 ABC 在内存上的表示方式（在 UTF-16BE 的情况下）为 `0x00 0x41 0x00 0x42 0x00 0x43`，并不兼容现存的 ASCII 编码。

为了解决上述问题，出现了**UTF–8**。首先，Unicode 在 0x00~0x7F 的范围内分配了和 ASCII 编码相同的码位，由此，在 UTF-8 中上述范围的字符可以用 1 个字节表示，在这之后的 0x80~0x7FF 用 2 个字节的 0xC280~0xDFBF 表示，0x800~0xFFFF 用 3 个字节的 0xE0A080~0xEFBFBF 表示。用语言难以描述清楚，还是看图 5-1 吧。

图 5-1
UTF-8 的二进制表示方法

```
到第 7 位为止   0vvvvvvv
到第 11 位为止  110vvvvv 10vvvvvv
到第 16 位为止  1110vvvv 10vvvvvv 10vvvvvv
到第 21 位为止  11110vvv 10vvvvvv 10vvvvvv 10vvvvvv
到第 26 位为止  111110vv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv
到第 31 位为止  1111110v 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv
```

※ 本图中的“v”表示字符编码的位都存储在靠右的位置。

在图 5-1 中的“V”表示字符编码的位都存储在靠右的位置。这种方法的好处在于，在只使用 ASCII 字符的情况下兼容现存的 ASCII 编码，而且，由于 ASCII 字符以外的字符（UTF-8 需要 2 字节以上来表示的字符）全部使用 0x80 以上的字节依次表示，因此即便只考虑了 ASCII 编码的程序（编译器等），（只要能通过 8 位）也可以正常地处理 UTF-8 的文件。像 GB2312 中 0x5C 那样的问题不会在 UTF-8 中出现。同样，在 GB2312 中有搜索“海”却被匹配成“""”的问题（因为“海”的 GB2312 编码为 0xB0 0xA1，“""”的编码为 0xA1 0xB0），UTF-8 的第 1 个字节不会与其他字符的第 2 个以及之后的字节重复，因此不会有问题。另外，UTF-8 的表示方式是以字节为单位的，所以也不会受字节序的影响 *。

* 因此从逻辑上来讲不再需要 BOM 了，但还是有附加了 BOM 的编辑器和没有 BOM 就不能正常运行的应用存在。

UTF-8 的缺点在于，用 UTF-16 的 2 个字节可以表示的字符，在 UTF-8 中需要 3 个字节才能表示。

话说回来，图 5-1 中所示，UTF-8 最大为 6 字节，可以表示 31 位的码位（实际分配到了 4 字节 21 位）。但是，在使用 UTF-16 的情况下连表示 21 位的字符也做不到。因此需要使用一种称为代理对（Surrogate Pair）的方法，即如果最初的 2 个字节在特定范围（0xD800~0xDBFF）的话就要连接后面的 2 个字节来表示 1 个字符，以此方法表示 0x10000~0x10FFFF 之间的字符。0x110000 以后不能用

UTF-16 表示。

补充知识 Unicode 可以固定（字节）长度吗?

就像 5.1.2 节介绍的那样，在内存中表示某个字符串的时候，如果每个字符占的内存空间大小是可变的，就会很不方便。试想一下，只要写 `str[i]` 就能取到 `str` 的第 `i` 个字符的话确实很方便。

基于这点，在 Unicode 中 UCS2 范围内所有的字符都可以用 2 个字节表示，要是能忍受，即使 ASCII 字符也要消耗 2 个字节的话就太完美了。想法总是美好的，结果接下来发现 2 个字节容不下了，又引入了代理对，结果 1 个字符又失去了固定长度。真是蠢死了。肯定会有人这样想（实际上我以前就是这么想的）。

但是，Unicode 在最初的建议稿（1989 年 9 月的 Unicode Draft1）中就提出了，以在普通的罗马字后面加上方言记号的形式（合成字符）表示德语的元音变音及类似的字符。因此，Ä方言在 Unicode 中表示为 `U+0041 U+0308`（但是为了与现存的 Latin-1 编码兼容，也可以用 `U+00C4` 表示）。

总之，Unicode 从一开始就没有想让一个字符用固定长度表示。

5.3 crowbar book_ver.0.3 的实现

本节将要说明如何在 crowbar_ver.0.3 中实现（实现到什么程度，怎么实现）对中文的支持。

5.3.1 要实现到什么程度?

crowbar 对中文（或者说国际化）的支持应该到什么程度呢？这个“程度”包含了多方面的含义。首先，到目前为止，crowbar 的变量或者函数名之类的标识符只支持字母。

Java 等语言可以使用汉字命名变量，但我想很少有人会用到（这只是因为习惯的问题，其实用汉字来命名变量，代码可读性没准会有飞跃性的提高），这个功能即使支持了中文也没人会用到。另外，如果要让标识符支持汉字，就要决定是否允许变量名里面包含全角空格。因为比较麻烦，这里就先跳过了。

另外还有一个问题就是，要支持一个什么样的字符集？

姑且以宽字符（`wchar_t`）为一个字符来处理。使用5.1.3节中介绍过的`mbtowc()`系列函数将多字节字符转换为宽字符。

在Linux中使用`sizeof(wchar_t)`返回4，但在Windows中返回2。因此，宽字符可以正常处理ASCII字符和普通的中文，但是超过了UCS2范围的字符（在UTF-16中被组成代理对的字符）就不能直接处理了。A方言这样的合成字符也是不能处理的（使用兼容Latin-1的U+00C4来表示就另当别论了）。当然并不是完全不能表示，如果使用字符串的`length()`方法获取字符串的长度的话，会得到和实际长度不同的结果。

这样一来，也不能说是完全支持了Unicode，但是对于大多数人来说，暂且让我可以正常地使用中文和英语就可以了。我想比起花很多时间来追求完美的支持，把大多数人觉得“可以”的范围赶快做出来，是更好的选择。

以下两种编码方式为用于输入输出的多字节的字符编码方式。

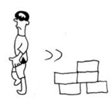
- GB2312
- UTF-8

先假设crowbar处理器中的C代码和用crowbar书写的代码以及`fgets()`等读取输入输出文件的函数，全部使用了统一的编码方式。

5.3.2 发起转换的时机

在crowbar中，以下这些情况需要由多字节字符转换为宽字符，或者由宽字符转换为多字节字符。

1. 用crowbar书写的代码中的字符串字面量在编译时需要转换为宽字符串。
2. `fgets()`函数读取的字符串需要由多字节字符串转换为宽字符串。
3. 调用`print()`、`fputs()`等输出函数的时候,需要由宽字符串转换为多字节字符串。
4. 因为C代码中使用GB2312（EUC-CN）嵌入错误信息，所以在组装错误信息时需要转换为宽字符串（信息在显示的时候，需要根据规则3再次进行转换）。
5. 在接收命令行参数`ARGS`的时候，需要转换为宽字符串。



5.3.3 关于区域设置

在 5.3.1 节中说道：

先假设 crowbar 的处理器中的 C 代码和用 crowbar 书写的代码以及 `fgets()` 等读取输入输出文件的函数，全部使用了统一的编码方式。

可是，编码方式究竟是什么呢？简单地说，就是环境默认的字符编码。Windows 是 GB2312，Linux 是 EUC-CN 或者 UTF-8。UNIX 可以根据环境变量 `LANG` 进行切换。

那么，实际上使用 `mbtowc()` 系列函数将多字节字符串转换为宽字符串，想要为转换函数群指定默认区域设置必须调用下面的函数。

```
setlocale(LC_CTYPE, "")
```

在 crowbar 中，需要在 `main()` 函数中执行上面的语句。

`setlocale()` 函数的详细设计在这里不再赘述，请参考（C 语言标准库的）手册等资料。

5.3.4 解决 0x5C 问题

在 5.1.1 节中提到，当前的 crowbar 由于运行在 GB2312 环境下，字符串字面量中不能使用例如“昞”这样的字符。

即使可以使用 `mbtowc()` 系列函数正确地处理 GB2312，还必须在一开始就用 lex 解释字符串字面量，这还不算完，为了解决这个问题还要在 crowbar.l 中添加代码。

GB2312 的汉字，第 1 个字节定义在 0xA1-0xF7 之间，第 2 个字节定义在 0xA1-0xFE 之间。因此，如果只支持 GB2312 的话，要在 crowbar.l 中添加下面 4 行代码。

```
（之前省略）
<STRING_LITERAL_STATE>[\xa1-\xf7][\xa1-\xfe] {
  crb_add_string_literal(yytext[0]);
  crb_add_string_literal(yytext[1]);
}
（之后省略）
```

为了在编译时能区分源文件的编码，在解释器中保存一个标识。

```
/* 保存编码方式的枚举类型 */
typeof enum {
    GB_2312_ENCODING=1, /* GB2312 */
    UTF_8_ENCODING      /* UTF-8 */
} Encoding;

struct CRB_Interpreter_tag {
    （中间省略）
    /* 在 CRB_Interpreter 结构体中保存
        crowbar 代码的编码方式 */
    Encoding    source_encoding;
};
```

在此基础上，在 lex 的启动条件中添加 GB_2312_2ND_CHAR（GB2312 的第 2 个字节），代码在读取到了 GB2312 的第 1 个字节时跳转到 GB_2312_2ND_CHAR 执行。

```
<STRING_LITERAL_STATE>.     {
    /* 从解释器中取得编码方式 */
    Encoding enc = crb_get_current_interpreter()->source_encoding;
    /* 先将字符添加到字符串字面量中 */
    crb_add_string_literal(yytext[0]);
    /* 如果代码运行在 GB2312 环境下，
       再判断这个字符，如果是 GB2312 的第 1 个字节，
       跳转到 GB_2312_2ND_CHAR 执行 */
    if (enc == GB_2312_ENCODING
        && ((unsigned char*)yytext)[0] >= 0xa1)
        && ((unsigned char*)yytext)[0] <= 0xf7)) {
        BEGIN GB_2312_2ND_CHAR;
    }
}

<GB_2312_2ND_CHAR>. {
    /* 添加 GB2312 的第 2 个字节 */
    crb_add_string_literal(yytext[0]);
    BEGIN STRING_LITERAL_STATE;
}
```

增加 CRB_Interpreter 的 source_encoding 成员，是因为在创建 CRB_Interpreter 的内存空间时，不能使用 #ifdef 进行分割（请参考 interface.c 的函数 CRB_Interpreter()）。

补充知识　失败的#ifdef

如前面所述，在执行解释器的处理器中，用 #ifdef 来切换语言的（默认）设定（GB2312、GBK 或者是 UTF-8）。在最初的 crowbar 中，就是使用 #ifdef 进行对应处理器的切换。

在一些 C 的入门书中都有这样一句话：为了提高移植性而适当地使用 #ifdef。以我的理解，“适当地使用”其实就是“尽量别用”的意思。因此，这次我（在处理器切换时）使用了 #ifdef，这对我来说也是一次失败。

根据处理器不同而使用 #ifdef 选择不同代码片段的话，会使代码变得很难理解。另外，像这样分散的代码通常很难进行充分的测试。在理想状态下，所有 #ifdef 的组合可以伴随着每日构建进行自动化测试，这感觉还不错，但是我认为这在实际中很难实现。

如果是为了提高移植性，那么也可以不使用 #ifdef 来处理各种分支，只要写一个尽可能适应各种处理器的代码不是就行了吗？

编程方面的著作《程序设计实践》[5] 中有以下记载。

> **如果我们对于条件编译持否定态度，那么就会由此发生一些问题。先不说最麻烦的。条件编译基本上都不可能进行测试。（中间省略）在对其中一个 #ifdef 代码块进行测试的同时，如果想测试另外的 #ifdef 代码块，除非改变环境使另一个 #ifdef 代码块生效，否则无法进行验证。**
>
> **（中间省略）**
>
> **由此我们得知，让我们感兴趣的是，在所有目标环境中都可以运行的共通性功能。**

5.3.5 应该是什么样子

5.3.1 节决定了 crowbar 不处理合成字符和 UCS2 范围以外的字符。

如果只是为了对应中文的话，这样的设计（指 5.3.1 节中提到的设计方式）就没问题了。但如果想要完美地实现，恐怕就需要考虑以下几点（以 Unicode 为前提）。

1. 内部表现也要使用 UTF-8

如果考虑合成字符的话，就不可能让字符有固定长度。如果想要取得字符串的第 n 个字符，每次都必须从字符串的开头扫描，所以还是算了吧。

2. 不使用mbtowc()系列函数，自己实现全部的转换

如果自己保存转码表，就要根据不同的情况使用不同的转码表。比如，在需要和 Java 兼容的时候要使用 Java 的转码表，如果要在 Windows 对话框中显示一个字符串的时候又要使用 Windows 的转码表等。

mbtowc()系列函数不仅意味着“在所有的处理器中，总是可以返回所期望结果”，还表示“如果自己保存转码表的话，所有转换都要自己进行”。

作为一个还算现实的做法（只要能处理好中文就可以了），我制作的这个语言处理器，正好解决了所有的问题。如果一味追求结果而不能实现也是没有意义的。

补充知识　还可以是别的样子——Code Set Independent

在 5.3.5 节中介绍了自己保存转码表和将内部编码变为 UTF-8 这两种方法。

在 UTF-8 这种方法中，首先让内部的编码方式使用 Unicode，在正常的情况下，不论是从外部输入的字符编码还是向外部输出的字符编码都是 Unicode。

除此之外还有另外一种方式，即内部编码不固定。这种方式称为 Code Set Independent（CSI）。

若将内部编码固定为 Unicode，那么在 UNIX 的 EUC 环境中，是绝对不可能使用 EUC 以外的编码方式的。在这种情况下，每次发生读写时都要使用转码表在其中转换。暂且不说效率低下的问题，更重要的是，如果想要表示在 Unicode 中没有的字符，或者要把在 Unicode 中认为是一样的字符当做不同的字符处理，这些情况使用 Unicode 都是不能处理的。

实际上，Ruby1.9 就采用了 CSI 方式。在 Ruby 中，每个字符串都会保存着自己的编码方式。

例如在输出文件的情况下，只要 Ruby 知道转换方法，就可以将要输出的编码方式（外部编码方式）和字符串的编码方式（内部编码方式）进行转换。

具体来说，比如在 Ruby 支持的编码方式中有 Emacs-Mule，这种编码方式没有采用像 Unicode 一样将中文汉字分配统一编码的方式，它基于 ISO-2022 为各国（但是没有国籍限制）语言分配了不同的编码[①]。在处理以这种方式编码（同时存在多种语种的文字）的文件时，如果像 crowbar 那样限定内部字符编码为 Unicode 的编码方式，那么在转换为内部表现时就会产生不可逆的（无法恢复到原来状态的）信息丢失。

CSI 既有优点也有缺点。

当有 N 种外部编码方式时，Unicode 正常的处理方法是准备输入和输出的编码方式转换器（共 2N 种）。与此相对，CSI 只需要（N-1）种。另外，在程序中进行比较和连接字符串的时候也会受到限制。

现在，为了方便实现而优先将内部编码限定为 Unicode，这实际上还是有些问题的，这就是我坚持 CSI 的原因。

这是一个哲学或者说是价值观的问题。两种方式都有它们的合理性，在使用的时候应该在平衡利弊的基础上再做出决定。

① 只是处理了不同的语种，但不是按国家划分的编码。——译者注



第 6 章

制作静态类型的语言 Diksam

6.1 制作 Diksam Ver 0.1 语言的基本部分

此前的章节中，我们已经制作了一个无变量类型、通过分析树执行的语言 crowbar。从本章开始，我们来制作一个静态类型并通过字节码执行的语言 Diksam。

Diksam 这个名字的由来可不是“像 Perl 一样”的双关语，而是我钟爱的一种红茶的名字。既然有了咖啡语言 Java，再来一个红茶语言也无伤大雅吧。Diksam 是一种口味浓郁的阿萨姆红茶，一般都用来做成奶茶。我很喜欢直接去感受它的那份浓郁。

6.1.1 Diksam 的运行状态

前面已经提到，Diksam 是通过字节码执行的语言。

说起通过字节码执行的语言，以 Java 为例再合适不过了 *。在编写 Java 代码时，要先编写源程序，再通过 javac 编译成保存着字节码的class 文件。

* 虽然现在可能基本上都以 JIT（Just In Time）的方式运行，但是本书中不考虑这种情况。

Diksam 语言不会生成像 class 文件那样的字节码文件。编译器解析源代码并生成分析树后，直接在内存中生成字节码。这些在内存中生成的字节码，会由 Diksam 的虚拟机 DVM（Diksam Virtual Machine）运行。

Diksam 不生成字节码文件的做法，免去了考虑文件格式和文件编码的麻烦。当然，除了我的个人考量外，对于用户来说也省去了逐个编译源文件的工夫。这种做法在代码量非常巨大的情况下，每次启动程序时花费在编译上的时间都会让你等得不耐烦。当然，程序不是很大的时候还是很方便的。

另外，在实现中，DVM 将完全信任编译器生成的字节码。所谓“完全信任”就是说，即使是含有恶意代码的字节码 DVM 也会加载执行，不过这可能会使 DVM 崩溃。

Diksam 是一种静态类型语言，在编译时就已经决定了所有变量和表达式的数据类型。因此，在运行时就不需要再检查变量的数据类型了。比如，为了在堆中存储一个字符串，string 型变量会（间接的）保存指向该字符串的指针，但如果存储的整数型局部变量被当作 string 型引用时，DVM 就会因为找不到正确的引用地址而崩溃。

但也是有相应对策的，比如在网页中运行的 Java Applet，由于必须执行来自外部的（不能被信任的）字节码，因此绝对不允许发生上述情况。所以 Java 在加载字节码的时候会执行**验证器**（verifier）程序，以验证加载的字节码是否正确。

但是，制作验证器非常复杂，而 Diksam 还只是直接在内存中执行字节码的语言，所以……不好意思，这里我要跳过啦。

6.1.2 什么是 Diksam

首先，我们制作的 Diksam 是 Diksam book_ver.0.1 版本，实例请参考代码清单 6-1。

代码清单 6-1
fizzbuzz_0_1.dkm

```
 int print(string str);

int i;
for(i = 1; i <= 100; i++){
    if(i % 15 == 0){
       print("FizzBuzz\n");
    }elsif(i % 3 == 0){
       print("Fizz\n");
    }elsif(i % 5 == 0){
       print("Buzz\n");
    }else{
       print("" + i + "\n");
    }
}
```

6.1.3 程序结构

与 crowbar 相同，Diksam 中也有顶层结构，并且允许在函数外编写代码。程序从顶层结构的开头开始执行。

Diksam 用下面的方式定义函数。与 C 语言基本相同。

```
int func(int a, double b){
    int local_varible;// 声明局部变量

    local_varible = a + b;
```

```
    print("local_varible.." + local_varible + "\n");

    return local_varible;
}
```

6.1.4 数据类型

Diksam book_ver.0.1 可以使用以下四种数据类型。

- boolean（布尔）型。可以赋 true 或 false 值。
- int（数字）型。
- double（浮点）型。int 和 double 混合进行运算时，参与运算的 int 类型会自动扩展为 double 类型。
- string（字符串）型。当字符串在 + 号左边时，右边的部分会自动转换为 string 型。

double 这个关键字在 C 语言中的意思是**双精度**浮点数，这里的 double 以 float（单精度浮点数）的存在为前提。但是，Diksam 中并没有 float，尽管它叫作 double 型，也纯粹是为了照顾 C 和 Java 语言的使用者而已。

6.1.5 变量

如前所述，Diksam 是静态类型的语言，变量必须事先声明，声明方式和 C 语言一样，如下所示：

```
int a;
```

在函数内声明的局部变量只可以在声明变量的程序块内引用，函数外声明的变量是全局变量[①]。

现阶段在 Diksam 中变量声明也是一种语句，可以写语句的地方就可以声明变量。

在 C 语言中变量必须声明在程序块的开头部分，C++ 和 Java 取消了这个限制。虽然这被大多数人认为是个改进，而且 Diksam 也是这么做的，但是老实说我不喜欢这个改进。我认为，在代码的任意位置都可以声明变量会使一个函数渐

① 程序块内外都可引用。——译者注

渐变得冗长。因此，也许可以趁现在[1]改正过来。

Diksam 中声明变量的方式和 C 等语言相同，采用类型 变量名 ; 的形式。这种形式对于处理数组和函数类型的变量来说很麻烦。如果引入预声明关键字 var，那么像下面这样把数据类型写在后面的语法（Pascal、Ada、ActionScript 都是这种方式）处理起来会简单一点。

```
var a:int;
```

但是，习惯了 C 或者 Java 语言的程序员肯定会说，还是类型 变量名 ; 这样的方式写着更顺手。为了照顾大家，Diksam 还是采用了大家比较习惯的方式。实际上，在拙著《征服 C 指针》（人民邮电出版社，吴雅明译）中，整本书都在表达对 C 语言变量声明语法结构的不满。在那本书中写了那么多不满，现在却变成了墙头草，肯定要被嘲笑了。

言归正传，类型 变量名 ; 形式的语法如果包含了数组和类（class），制作语法分析器的时候就会出现各种问题，幸亏 Diksam 在语法规则上花了不少功夫才避免了这些问题。这个话题我们在后面的章节中详细说明。

变量**初始化语句**（initializer）的方式也与 C 语言相同。

```
int a = 10;
```

6.1.6 语句和流程控制

流程控制语句有：`if`（`elseif`、`else`）、`while`、`for`、`break`、`continue`、`return`。

含义和 crowbar 相同，花括号也不可以省略。另外，`break` 和 `continue` 也可以使用标签（和 Java 一样）。

6.1.7 表达式

Diksam 中可以使用的运算符及其优先级，如表 6-1 所示。

① 制作 Diksam 语言的机会。——译者注

表 6-1 在 Diksam 中可以使用的运算符

<table>
<tr><th>运算符</th><th>含义</th></tr>
<tr><td><code>++ -- ()</code></td><td>自增、自减、函数调用</td></tr>
<tr><td>（单目）-</td><td>符号反转</td></tr>
<tr><td><code>* / %</code></td><td>乘法、除法、模</td></tr>
<tr><td><code>+ -</code></td><td>加法、减法</td></tr>
<tr><td><code>< <= > >=</code></td><td>比较大小。字符串也可以比较大小（以 strcmp() 为基准）</td></tr>
<tr><td><code>== !=</code></td><td>等值比较。字符串也可以进行比较（比较的不是引用而是值）</td></tr>
<tr><td><code>&&</code></td><td>逻辑与（AND）运算符。短路运算符，在表达式 a && b 中，如果 a 为真，b 就不判断了</td></tr>
<tr><td><code>||</code></td><td>逻辑或(OR)运算符。短路运算符，在表达式 a || b 中，如果 a 为真，b 就不判断了</td></tr>
<tr><td><code>= += -= *= /= %=</code></td><td>赋值运算符。与 C 语言中的含义相同</td></tr>
<tr><td><code>,</code></td><td>逗号运算符。从左到右的顺序计算表达式，返回右边表达式的值</td></tr>
</table>

函数调用被定义为运算符，说明函数也是一种表达式。

这就意味着，在 crowbar 中可以像 C 语言的函数指针那样把函数赋值给变量，但是，实际上 crowbar 并不能声明保存函数的变量，因此也就没办法把函数赋值给变量。

6.1.8 内建函数

Diksam 在一开始就准备了内建函数。Diksam book_ver.0.1 中的内建函数一览表如下所示。

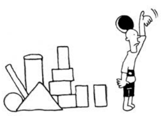
int print (string arg)	显示 arg。返回值为 0（因为还没有 void 类型）

完毕！

只有一行的一览表，很糗是吧。

6.1.9 其他

注释可以使用 C 风格的 `/*~*/`，也可以使用 C++ 从 `//` 开始到本行结束的风格。



6.2 什么是静态的 / 执行字节码的语言

6.2.1 静态类型的语言

如前所述，Diksam 是静态类型的执行字节码的语言。这样的语言有以下优点。

1. 能够高速执行（通常情况下）

理由会在之后详述。对于静态类型的执行字节码的语言来说，如果能够朴素地实现这两个方面的话，会比无类型并通过分析树执行的语言拥有更快的速度。

2. 编译阶段和执行阶段分离

例如在 Java 中，事先使用 javac 把源代码生成为 class 文件，这样在执行时就不需要再编译了。

实际上 crowbar 由源代码生成分析树的处理并不耗时，（Diksam 要想比 crowbar 强的话）速度上没有什么可期待的。相反，如果想要在不发布源代码的情况下运行程序倒还是有希望的。话虽如此，实际上因为现在的 Diksam 是在内存上构建字节码并执行的，结果还是必须要发布源代码。

3. 相比之下使用静态类型的语言编写的代码更易读

静态类型的语言通常要将变量和类型一起声明（比如 `int a;`）。也许会有人觉得这样很麻烦（这就是为什么无类型的 LL 语言这么有人气的原因），实际上也不是很费事，因为还是把变量类型明确化以提高源代码的可读性更为重要。至少我是这么想的，但这句话是一个泥潭，还是赶快结束这个话题吧。

6.2.2 什么是字节码

字节码是在被称为**虚拟机**（Virtual Machine）的虚拟 CPU 上执行的机器语言。机器语言直接通过 CPU 执行，而字节码通过虚拟机执行。

字节码和机器语言一样，实际上都是数字的序列（这点也和机器语言一样）。

为了让大家更好的理解，各个命令将以一种被称为**助记符**（mnemonic）的字符串形式表现出来。

比如在 Java 虚拟机（JVM）中运行的字节码就是下面这个样子。（再来回看一下代码清单 1-4）

```
 0: bipush 10
 2: istore_1
 3: iload_1
 4: bipush  10
 6: if_icmpne     20
 9: getstatic
12: ldc
14: invokevirtual
17: goto    28
20: getstatic
23: ldc
25: invokevirtual
```

可能有人会认为，就算是为了让读者能够更好地理解才使用了字符串的表现方式，可这种根本就不知所云的东西如果想要从源代码生成出来，简直就是做梦，更不要说作一个可以生成字节码的编译器了。但实际上并没有那么难，具体步骤请见下节。

6.2.3 将表达式转换为字节码

crowbar 从 ver.0.2 版本开始把计算过程中的值存入栈中（为了确保 GC 可以追踪指针）。在这种情况下，执行字节码的语言也会进行大致同样的实现。关于此事，4.2.3 节的最后部分中有如下记载。

这种做法与 JVM 堆栈机运行字节码时的栈动作相同。如此一来，我们就向字节码解释器又迈进了一步。

具体来说，分析树优先从最深层次开始遍历，从而生成下面这样的代码。

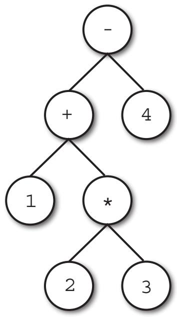
1. 常量/变量类会直接把值保存到栈中。
2. 双目运算符以先左后右的顺序保存到栈上，从栈顶端的两个元素开始计算，并将结果保存到栈中。

比如：

```
1 + 2 * 3 - 4
```

这个表达式会展开成图 6-1 所示的分析树。

图 6-1
分析树

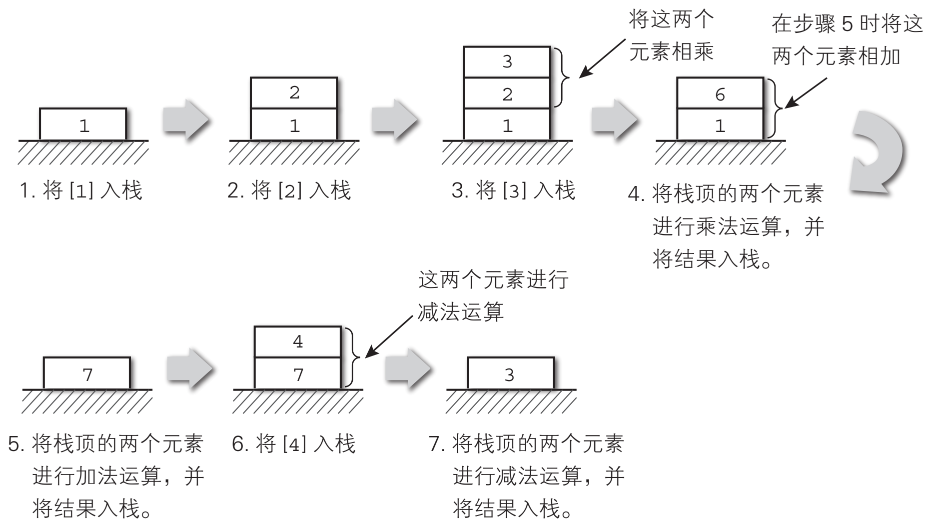


这个分析树以自上而下的顺序遍历，同时，常量的节点会生成字节码 push 值，运算符节点生成代表运算符的字节码，生成的字节码如下所示。（下面的代码只是用作说明的模拟代码。这些字节码既不是 Java 的，也不是 Diksam 的。）

```
1: push 1    # 将 [1] 入栈
2: push 2    # 将 [2] 入栈
3: push 3    # 将 [3] 入栈
4: mul       # 将栈顶的两个元素进行乘法运算，并将结果入栈。
5: add       # 将栈顶的两个元素进行加法运算，并将结果入栈。
6: push 4    # 将 [4] 入栈
7: sub       # 将栈顶的两个元素进行减法运算，并将结果入栈。
```

此时栈的动作如图 6-2 所示。

图 6-2
执行字节码时的栈动作



在表达式中不止有上例中的双目运算符，还有单目运算符，而且它们的思路是一样的。比如单目运算符的减号，在栈中执行的操作是“将栈顶的值取出，反转符号再存入栈中”。

但是，上述例子中只处理了整数。在实际的编程语言中还必须要处理实数，也有可能会出现“整数和实数相加”这样的混合运算。在这种情况下，必须要将参与运算的整数转换为实数再进行加法运算。在有些语言中，字符串和整数也可以进行加法运算 *。接下来的操作就变成了将整数转换为字符串然后再将其连接。

* 有些语言使用 +，有些使用 . 作为运算符。顺便说一下，在 crowbar 和 Diksam 中使用的都是 +。

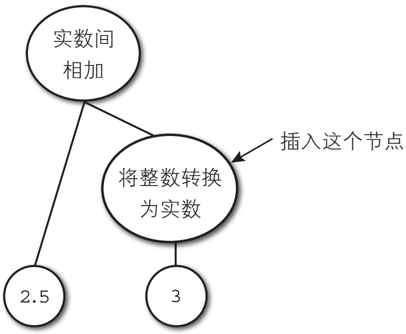
在执行这样的操作时，有两种方法。

第一种是在向栈中保存值的时候就加入能够区分类型的标识，在运行时再进行判断。crowbar 使用的就是这个方法。`CRB_Value` 结构体的成员 `type` 保存的就是类型的标识，在运行时根据这个标识进行转型和运算。

另外一种方法是，在编译时进行类型判断。

如果有必要转型的话，类型转换处理将在编译时进行。那么，像图 6-3 这样一个分析树就能够实现类型的转换了。

图 6-3
类型转换的分析树



这个分析树生成的字节码如下。

```
1: push_double  2.5
2: push_int     3
3: cast_int_to_double
4: add_double
```

第三行的 `cast_int_to_double` 命令将栈顶的 `int` 值转换为 `double`。

在这种方法中，如果是加法运算，究竟要如何区分是整数之间相加，还是实数之间相加或者是字符串连接呢？在编译完成之后就完全清楚了。因此，在执行

时无需加入多余的判断，也可以加快执行速度。

但是这样一来，在编译时就必须要知道变量的类型，因此必须要让用户进行带变量类型的**声明***。

* 静态类型的函数式语言，即使不声明类型，编译器也可以根据**类型推论**推测出来。

6.2.4 将控制结构转换为字节码

像 if 和 while 之类的控制结构，在字节码中使用 goto 这样的跳转（jump）命令实现。

比如下面这段 if 语句。

```
if (a == 10) {
    条件成立时的处理
    条件成立时的处理
}
后续的处理
```

将这段代码用字节码来表示。

```
1: push 变量 a 的值
2: push_int      10
3: eq_int             # 栈顶的两个 int 之间进行比较（==），并将结果入栈
4: jump_if_false 7    # 栈顶的值如果是 false 就跳转到第 7 行
5: 条件成立时的处理
6: 条件成立时的处理
7: 后续的处理          # 从第 4 行跳转到本行
```

在 crowbar 中，控制结构也是用分析树来表现的。程序在执行的同时对分析树进行递归，为了实现 break 和 continue 这样的语句，程序必须要从递归的最深层开始（使用特殊的返回值）。在执行字节码的语言中可以更简单地实现 break 和 continue，（如果必要的话）goto 也可以简单地实现（递归）。

6.2.5 函数的实现

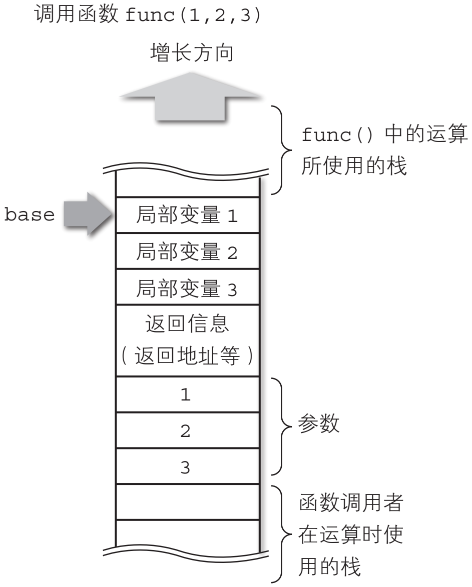
在 C 等语言中，通常情况下函数的调用也是用栈来实现的。

这种情况在拙著《征服 C 指针》中有详细的介绍，请各位读者参考。其中大致说明了在 C 的情况下，（简单实现的话）函数的调用遵循以下步骤（如

图 6-4)。

1. 将参数（从后面开始）入栈
2. 将返回地址等返回信息入栈
3. 将局部变量所占内存区域入栈

图 6-4
语言中的函数调用



C 语言的参数从后面开始入栈是为了实现像 printf() 这样可变长参数的函数。这次制作的 Diksam 语言中没有可变长参数，因此没有必要特意从后面开始（向栈内）保存参数（实际上 Diksam 是从前面开始保存参数的）。

所谓返回地址，和字面意思一样，指向了函数终结时返回的地址。当函数从（栈的）某处开始调用之后，如果不将返回地址保存到栈中，在函数结束时就不知道要返回到哪里去了。

当所有局部变量保存到栈中的时候，栈的顶部就在图中 base 箭头所指的位置。利用以 base 为基准的偏移量，可以指定局部变量或者参数 *。

在之前的 if 语句字节码的例子中，如果把中文“push 变量 a 的值”写成字节码的话，应该像下面这样（当 a 是局部变量的情况下）。

```
push_stack_int       0 # 0 是变量 a 基于 base 的偏移量
```

在需要声明变量的语言中，全局变量也要在编译时全部决定下来，因此不能为这些全局变量指定偏移量的编号。比如 int 型的全局变量的值向栈中保存的字

* 另外，在机器语言中保存着以 base 为起点的偏移量引用地址的**寄存器**(register)，我们称其为**基址寄存器**(寄存器在机器语言中类似变量)。本书并不是关于机器语言的书，因此以后都用 base 来表示。

节码应该是下面这样。

```
push_static_int     0 # 0 是这个全局变量的索引
```

6.3 Diksam ver.0.1 的实现——编译篇

6.3.1 目录结构

Diksam 为了使编译器与执行器（DVM）分离，使用了以下的目录（directory）结构。

- compiler
  包含 Diksam 的编译器代码。函数名等的前缀为 `DKC`，`main()` 函数暂时放在这里面。
- dvm
  包含 Diksam 的执行器（DVM）代码。函数名等的前缀为 `DVM`。
- share
  包含 compiler、dvm 两个模块共享的代码。函数名等的前缀为 `dvm`。
  严格来说，这里打破了命名规则，这是因为考虑到是否要给只在编译器和 DVM 中使用的代码加一个公共的前缀，还有编译器也是依赖 DVM 的。
- include
  包含在多个目录中被引用的头文件。
- memory
  在介绍 crowbar 的时候，在 3.2.2 节中制作的内存管理的库。函数名等的前缀为 `MEM`。
- debug
  在介绍 crowbar 的时候，在 3.2.2 节中制作的用于调试的库。函数名等的前缀为 `DBG`。

在表 6-2 中描述了 include 中包含的头文件。

表 6-2
头文件一览表

头文件名	说明
DKC.h	Diksam 编译器库的公用头文件。用到了 Diksam 编译器的用户程序需要 `#include`
DVM.h	Diksam 虚拟机的公用头文件。用到了 Diksam 执行器的用户程序需要 #include
DVM_code.h	由 Diksam 编译生成的字节码对象。定义了 `DVM_Executable` 结构体的头文件。Diksam 编译器和 DVM 都会使用到这个文件
DVM_dev.h	头文件中定义了用于 Diksam 的编译器生成字节码的结构体 DVM_Executable。Diksam 的编译器和 DVM 都会使用这个文件
share.h	编译器和 DVM 的共享模块 share.o 的公用头文件
DBG.h	用于调试的模块 `DBG` 的公用头文件
MEM.h	内存管理模块 `MEM` 的公用头文件

6.3.2 编译的概要

编译按照下面的顺序进行。

1. **构建分析树**

   在 create.c 中实现。

2. **修正分析树**

   在这个阶段中进行的操作有，为表达式分析树中的各节点加入类型，如果存在不同类型间的运算时加入转换节点，将表达式中用到的变量与其声明绑定。

   上述操作会尽量在构建分析树的同时完成，实在不行的话也会在其他阶段中进行。此阶段（也就是所谓的“语义分析”阶段）会在 fix_tree.c 中执行。

3. **生成字节码**

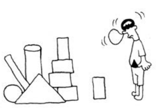
   以自上而下的顺序遍历分析树生成字节码。在 generate.c 中实现。

6.3.3 构建分析树（create.c）

构建分析树其实跟 crowbar 相比没有什么变化。如果一定要说有什么不同的地方，也只能讲讲“程序块”的处理了。

Diksam 局部变量的作用域在它声明的程序块中。

比如，表达式中使用了叫作 a 的变量时，编译器会首先探测最内层的程序块中是否声明了 a，如果没有，再逐个探测外层的程序块。为了实现这种处理方式，



在程序块的结构体中包含了 outer_block 成员。

```
typedef struct Block_tag {
    BlockType           type;
    struct Block_tag    *outer_block;    ←这个
    StatementList       *statement_list;
    DeclarationList     *declaration_list;
    union {
        StatementBlockInfo  statement;
        FunctionBlockInfo   function;
    } parent;
} Block;
```

outer_block 保存了外层程序块的指针。

为了设定 outer_block，在 Block 结构体创建实例的同时，必须要知道哪个是当前程序块（新创建的程序块的外层程序块）。因此，在编译器本体（DKC_Compiler）中保存了 current_block。

```
struct DKC_Compiler_tag {
    MEM_Storage         compile_storage;
    FunctionDefinition  *function_list;
    int                 function_count;
    DeclarationList     *declaration_list;
    StatementList       *statement_list;
    int                 current_line_number;
    Block               *current_block;     ←这个
    DKC_InputMode       input_mode;
    Encoding            source_encoding;
};
```

设定这个 current_block 的时机是在程序块开始的时候（解释器读取到 { 的时候）。

比如在 crowbar 中 crowbar.y 有如下代码。

```
block
        : LC statement_list RC
        {
            $$ = crb_create_block($2);
        }
        | LC RC
        {
            $$ = crb_create_block(NULL);
        }
        ;
```

这个方法在程序块结束的时候没有任何动作。

为了能够设置 `current_block`，我们要在 diksam.y 里的规则中插入动作。

```
/* 总之还是想要实现"LC statement_list RC" */
block
        : LC
        {
            $$ = dkc_open_block();
        }
          statement_list RC
        {
            $$ = dkc_close_block($<block>2, $3);
        }
        还是有"LC RC"的规则，这里省略了。
```

这样的动作被称为**嵌入动作**。

yacc 在处理嵌入动作的时候，会嵌入一个虚拟的目标。嵌入动作将被作为这个虚拟目标的动作进行处理。即为：

```
block
        : LC { 嵌入动作 } statement_list RC
        {
            程序块结束时触发的动作。
        }
        ;
```

上面这段代码，等同于下面的代码。

```
block
        : LC dummy_target statement_list RC
        {
            程序块结束时触发的动作。
        }
        ;
dummy_target
        :
        {
            嵌入动作
        }
        ;
```

虚拟目标部分并没有声明类型，（如果想要使用的话）嵌入动作中向 `$$` 设定的值，记作 `$<类型>序号`。前面写到的 diksam.y 中向 `dkc_close_block()` 传递的参数 `$<block>2`， `$3`，其中 `$<block>2` 就是嵌入动作中向 `$$` 设定的

值，statement_list 就变成 $3 了（由于加入了嵌入动作，向后移动了一个[①]）。

6.3.4 修正分析树（fix_tree.c）

在 fix_tree.c 中将会扫描 create.c 生成的分析树，并进行错误检查、添加数据类型、将表达式中的标识符与声明绑定这些操作。

下面的项目将要说明的是这些操作的具体内容。

■ 常量表达式的包装

这个操作在 crowbar 中已经存在了，Diksam 表达式中的常量表达式（像 24 * 60 这样的表达式）在编译时就会被包装为常量。

在现在的 Diksam 中，以数值的加减乘除和模、+ 进行的字符串连接、单目减号、比较、单目！为对象。

■ 为表达式添加类型

在 Diksam 表达式的分析树中，每个节点都保存着自己的类型。在 Diksam 编译器中定义了用来表示表达式的结构体（见下面代码，diksamc.h）。

```
struct Expression_tag {
    TypeSpecifier *type; ←这里保存类型
    ExpressionKind kind; ←用 kind 表示表达式的类别
    int line_number;
    union {              ←以联合体的形式保存类别对应的值
        DVM_Boolean             boolean_value;
        int                     int_value;
        double                  double_value;
        DVM_Char                *string_value;
        IdentifierExpression    identifier;
        CommaExpression         comma;
        AssignExpression        assign_expression;
        BinaryExpression        binary_expression;
        Expression              *minus_expression;
        Expression              *logical_not;
        FunctionCallExpression  function_call_expression;
        IncrementOrDecrement    int_dec;
        CastExpression          cast;
```

① 原来是 $2。——译者注

```
    } u;
};
```

比如 3 这样的整数值节点被定为 int 型，int 型变量的类型也被定为 int 型，但是像（1 + 0.5）这样的表达式就要同时分析表达式左边和右边的情况来适当地扩展类型。

看了 Expression 结构体的定义可能就会明白，这个结构体的 type 成员保存了表达式的类型，它所指向的结构体 TypeSpecifier 的定义如下（定义在 /include/DVM/DVM_code.h 中）。

```
struct TypeSpecifier_tag {
    DVM_BasicType   basic_type;
    TypeDerive  *derive;
};
```

枚举类型 DVM_BasicType 的定义如下所示。

这个枚举类型能够表示所有“基本类型”。

```
typedef enum {
    DVM_BOOLEAN_TYPE,
    DVM_INT_TYPE,
    DVM_DOUBLE_TYPE,
    DVM_STRING_TYPE
} DVM_BasicType;
```

另外一个成员 derive 以“派生类型”的方式表示，相关的定义如下（类型定义保存在 diksamc.h 中，但是在 DVM_code.h 中也有与其相似的定义，两者之间的关系将在后面的章节中介绍）。

```
typedef enum {
    FUNCTION_DERIVE
} DeriveTag;

typedef struct {
    ParameterList   *parameter_list;
} FunctionDerive;

typedef struct TypeDerive_tag {
    DeriveTag   tag;
    union {
        FunctionDerive function_d;
    } u;
    struct TypeDerive_tag   *next;
} TypeDerive;
```

```
struct TypeSpecifier_tag {
    DVM_BasicType   basic_type;
    TypeDerive  *derive;
};
```

枚举类型 DeriveTag 是一种派生类型的表现。在现在的 Diksam 中还没有数组之类的类型，目前存在的派生类型只有函数类型（FUNCTION_DERIVE）。函数（派生类型）的定义保存在 FunctionDerive 中，具体来说就是参数的类型信息。

到底什么是派生类型？这个话题在《征服 C 指针》中已经举例做了大致地说明 *。比如 Diksam 的 print() 函数，它的定义就成了下面这样。

* 在《征服数组和指针》一书中也做了介绍，可以访问下面的网址阅读。http://avnpc.com/pages/devlang#pointer

```
int print(string str);
```

此时，被称为 print 的标识符的类型就是“返回 int 的函数（参数为 string 类型）”*。在应用了函数调用运算符 () 后，可以把“函数（参数为 string 类型）”这部分看作是 int 型。

* 在 C 语言的情况下，函数名在表达式中会被转换为“指向函数的指针”，但是在 Diksam 中没有做这种费力不讨好的事。

因为 TypeDerive 包含了成员 next，所以这个派生类型可以用链表形式链接。因此，可以表示为“返回‘返回 int 的函数（参数为 string 类型）’的函数（无参数）”*。如果把数组也看作是派生类型的一种，那么可以表示为“返回‘int 的数组的数组’的函数（参数为 string 类型）”。

* 但是，现在还没有这样的语法结构支持这种类型的声明，所以无法使用。Diksam 最终会引入 delegate 类型，但与函数的派生无关。

说起来，现在还不能声明“函数型的变量”，也没有数组。因此，现阶段函数调用的语法规则如果像下面这样定义的话，就没有必要表现为派生类型了。但是因为在下一个版本中考虑到要引入数组，所以进行了如下实现。

```
primary_expression
    : IDENTIFIER LP parameter_list RP
    ;
```

■ **增加转换节点**

在为表达式添加类型的过程中，会加入如 6.2.3 节中说明的转换节点。当前编译时默认执行的类型转换如下所示。

- **双目运算中的类型转换**

  int 和 double 在运算时会将 int 转换为 double。

  还有，左边是 string 类型的运算时，会把右边转换为 string。

- **赋值时的类型转换**

  赋值时，会根据左边调整右边的类型（+= 之类的赋值运算符也是一样）。

  对于函数的实际参数以及参数的返回值在赋值时也进行同样的转换处理。

■ 函数内的变量声明

在处理 `int a;` 这样的声明语句时，在 create.c 的阶段会创建 `Declaration` 结构体，其定义如下。

```
typedef struct {
    char            *name;
    TypeSpecifier   *type;
    Expression      *initializer;
    int             variable_index;
    DVM_Boolean     is_local;
} Declaration;
```

在 create.c 的阶段，`Declaration` 结构体中设置了变量名（`name`）、类型（`type`）以及构造函数（`initializer`）。

由于声明是语句的一种，在 `Declaration` 结构体中以成员方式保存着 `Statement` 结构体的联合体，但是在 fix_tree.c 中就另当别论了。fix_tree.c 中将 `Declaration` 结构体链接成了一个链表。

```
/* 将 Declaration 结构体连成链表的结构体 */
typedef struct DeclarationList_tag {
    Declaration *declaration;
    struct DeclarationList_tag *next;
} DeclarationList;
```

在函数以外的变量声明通过链表 `DeclarationList` 保存在 `DKC_Compiler` 中。与此相对，在函数内声明的变量的作用范围是程序块，因此 `DeclarationList` 保存在程序块（即 Block）中。

在 6.3.3 节中讲过，表示程序块的 `Block` 结构体的声明，在其中的 `declaration_list` 中保存了当前程序块中的声明。

函数的形式参数由于可以被当做函数内的局部变量来使用，因此这些变量的声明被设置在了函数最外层的程序块中。接下来要加入的是局部变量。

与此同时，将对 `Declaration` 结构体中还没有设置的 `variable_index` 和 `is_local` 进行设置。

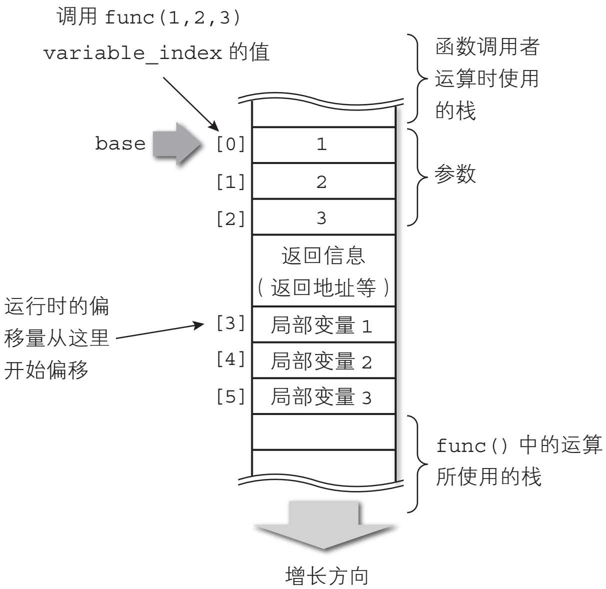
`is_local` 用来表示是否有局部变量的标识，在函数内声明变量的同时设置

其值，variable_index 为函数内声明的局部变量分配编号（最初的形式参数为 0）。

局部变量（包括参数）在栈中被创建，栈中的变量可以使用基于 base 的偏移量进行引用，具体请见 6.2.5 节。variable_index 就是这个“偏移量”，但是这里稍微有点复杂。

接下来我们看一下 Diksam 的栈，它实现了 DVM_Value 类型的数组，并且这个数组可以继续增大下标进行扩展。Diksam 此时的状态如图 6-5。

图 6-5
Diksam 的栈



在现阶段的编译器实现中，variable_index 从第一个形式参数开始顺序增长，其中并没有将返回信息考虑进去。因此，variable_index 和运行时的偏移量只能直接偏移到返回信息下面的局部变量。

当然，因为已知返回信息的字节大小，所以对于局部变量来说，加上它的大小后继续编号并非难事。但是，返回信息的字节大小依赖于 DVM 的实现，因此我想尽力避免将依赖于返回信息的值嵌入到字节码中。

最重要的是，现在的字节码只生成在内存中，并没有以文件等形式保存起来，因此即使嵌入了（返回信息的值），实际上也没有什么坏处。但如果将来要生成与 Java 的 class 文件类似的产出物时，可能就会出现问题。随着 DVM 版本升级，返回信息的字节大小也会随之发生变化，会出现之前编译的文件在新版

DVM 中不能运行的困扰。

于是，Diksam 编译器姑且先生成连续的编号，在执行前（加载 DVM 之后）再进行转换（请参考 6.4.1 节的小标题 3）。

话说回来，因为 Diksam 中变量的作用域是程序块，因此在下面这段代码生成字节码的时候，a 和 b 如果分配的是同一个内存空间（同一个 variable_index）的话，就可以节约栈空间了，因此在 Diksam 中的这种情况下会为它们单独生成索引。

```
if (a == 10) {
    int a;
     ⋮
}else {
    double b;
     ⋮
}
```

Diksam 会为没有初始化的局部变量决定取值，不会出现像 C 那样的不定值。在调用函数的时候，处理器会用 0 或者 null 之类的值为变量初始化。此时，虽然 a 和 b 都处于同一内存空间，但是由于类型不同，因此它们不能够用同样的位模型进行初始化。

■ 标识符和声明的绑定

在表达式中保存变量或者函数名的，是保存在 Expression 结构体中的联合体成员 IdentifierExpression 结构体。

```
typedef struct {
    char        *name;
    DVM_Boolean is_function;
    union {
        FunctionDefinition  *function;
        Declaration         *declaration;
    } u;
} IdentifierExpression;
```

这里的 function 或者 declaration 中存放的是函数定义或者变量声明。

在搜索局部变量的时候，会在与“当前程序块”相互连接的 Declaration 结构体中搜索，因此，ifx_xxx 系列的函数会把当前程序块作为参数传递进来。

6.3.5 Diksam 的运行形式——DVM_Executable

虽然总算能通过 fix_tree.c 生成字节码了，但是对于程序的运行来说不只需要字节码，还需要全局变量列表等必不可少的信息。在 Diksam 中定义了名为 DVM_Executable 的结构体用来保存字节码和刚才提到的那些相关信息，generate.c 的全部使命就是创建 DVM_Executable 结构体。

因此，首先要用下面这段代码来说明一下 DVM_Executable 结构体（DVM_code.h）。

```
struct DVM_Executable_tag {
    int                 constant_pool_count;
    DVM_ConstantPool    *constant_pool;
    int                 global_variable_count;
    DVM_Variable        *global_variable;
    int                 function_count;
    DVM_Function        *function;
    int                 code_size;
    DVM_Byte            *code;
    int                 line_number_size;
    DVM_LineNumber      *line_number;
    int                 need_stack_size;
};
```

如代码所示，利用可变长数组保存以下信息。

1. 常量池（constant_pool）

保存常量的内存空间。

2. 全局变量（global_variable）

保存全局变量列表。

3. 函数（function）

保存函数定义。将函数里要执行的语句的字节码保存在其内部（DVM_Function 结构体）。

4. 顶层结构的代码（code）

因为 Diksam 可以在顶层结构中书写代码，此成员用来保存这些代码生成的字节码。

5. 行号对应表（line_number）

保存字节码和与之对应的源代码的行号。

6. 栈的需要量（need_stack_size）

保存顶层结构的代码对栈的需要量。

每个函数对栈的需要量都保存在各自的 DVM_Function 中。

6.3.6 常量池

保存常量的内存空间被称为“常量池”。

比如，将 double 值 2.5 入栈的时候，Diksam 的字节码表示为：

```
push_double 2.5
```

在实际输出字节码的时候，命令 push_double 应该会被分配到某个代号（编号）。在 DVM 中会分配到十进制的 6，因此，6 被作为一个字节输出到字节码中。那么 2.5 该怎么办呢？在我的环境中 double 占 8 个字节，因此在 6 后面应该紧接着输出这 8 个字节。

老实说，因为现在的 Diksam 只在内存中保存字节码，编译器环境中 double 型的字节表现可以直接输出到字节码中。但是，如果将字节码输出到文件的时候就会出问题了。这是因为执行字节码的机器和编译字节码的机器可能会用不同的形式表示 double 型数据[1]。

“字节码中以某种正规化的表现方式进行保存，读取的时候再进行转换。”在进行这个处理的时候，如果字节码中间突然出来一个要转换的 2.5，那么处理起来会很麻烦。另外，不止是实数，字符串也是一样，如果在字节码中突然出现了一个嵌入的“hello, world\n”，这在普通的程序员看来也没那么美观吧。

因此我们在这里使用了常量池。常量池数组中的各个元素组成了下面的结构体。

```
typedef enum {
    DVM_CONSTANT_INT,
    DVM_CONSTANT_DOUBLE,
    DVM_CONSTANT_STRING
} DVM_ConstantPoolTag;

typedef struct {
    DVM_ConstantPoolTag tag;
    union {
```

① 从而导致解释字节码的方式也不同。——译者注

```
        int       c_int;
        double    c_double;
        DVM_Char *c_string;
    } u;
} DVM_ConstantPool;
```

如上面代码所示，把保存 int、double 或者字符串的成员定义为一个联合体。

在 Diksam 的字节码中，下列常量不会被嵌入到其中，而是保存到常量池。字节码中只保存常量池中对应的索引值。

- 负数或者 65536 以上的整数
- 0 或者 1 以外的实数
- 字符串

1~2 字节的整数以及实数 0 或者 1 使用下列命令进行处理。

- push_int_1byte
  在字节码上用这个命令将其后的 1 个字节作为整数保存。
- push_int_2byte
  在字节码上用这个命令将其后的两个字节作为整数并以大尾序保存。
- push_double_0
  实数运算中出现 0 的时候使用此命令（这里效仿了 JVM）。
- push_double_1
  实数运算中出现 1 的时候使用此命令（这里也效仿了 JVM）。

前面提到过字节码中只保存着常量池中用来引用常量的索引值，因为 1 个字节存不下这个索引值，所以现在 Diksam 在实现时使用了两个字节，如果使用 push_double 这样的命令会将其后的两个字节的整数值以大尾序保存起来。可是，两个字节是否就够用了呢？这是最大的悬念。实际上这里也是在效仿 JVM，我想可能修正一下这里会更好。

另外，相同的常量在程序中多次出现的时候，虽然在常量池上分配同样的入口可以节约常量池的内存空间，但是现在的 Diksam 没有这么做。这里照例只是偷了个懒而已。

补充知识　YARV 的情况

Diksam 将一部分常量的值保存到了常量池中。这个结构体实际上是效仿 JVM，但是在 Ruby 的 VM，也就是 YARV（Yet Another Ruby VM）中，就把常量值嵌入到了字

节码中。

这么做的理由是，像常量池这样把常量值保存在其他地方并通过配置索引指定操作数的方法，可能会对性能产生不利的影响[6]（因为是间接访问）。

还有，在 YARV 中的指令不止 1 字节，在处理器中为其分配了 1 个 int 的大小（这意味着 YARV 的指令不是一个严格的“字节码”）。虽然可惜了这些内存，但是对速度还是非常有利的。

Diksam 如果真的考虑性能的话，也许应该向 YARV 学习一下 *。

* Java 当初的使用方法被假设为是从网络下载的小应用程序，因此字节码是个必然的选择。

6.3.7 全局变量

DVM_Executable 结构体的 global_variable 成员，就如同其字面的含义表示的是全局变量。从字节码引用全局变量的时候，就会使用到这个 DVM_Variable 型的数组的索引。

比如，将整数型的全局变量的值 push 到栈中的命令是 push_static_int。

使用上述命令，将其后的两个字节以大尾序存入数组中并延续索引编号。

DVM_Variable 的定义如下。

```
typedef struct {
    char                *name;
    DVM_TypeSpecifier   *type;
} DVM_Variable;
```

一目了然，这里表示的是全局变量的名称和类型。

全局变量的类型是为了在开始运行的时候进行初始化以及垃圾回收时使用的。

变量名到目前为止还没有用到。前面也提到过，利用索引从字节码中引用全局变量。但是我想，在实现调试器的时候，变量名是一个必要的信息。

另外，使用 DVM_TypeSpecifier 结构体来保存全局变量的类型。在编译时类型信息被保存到类型信息 TypeSpecifier 结构体中。也就是说，在 generate.c 中实现从 TypeSpecifier 结构体到 DVM_TypeSpecifier 结构体的复制。

6.3.8 函数

表示函数的结构体 DVM_Function，其定义如下所示。

```
typedef struct {
    DVM_TypeSpecifier   *type;
    char                *name;
    int                 parameter_count;
    DVM_LocalVariable   *parameter;
    DVM_Boolean         is_implemented;
    int                 local_variable_count;
    DVM_LocalVariable   *local_variable;
    int                 code_size;
    DVM_Byte            *code;
    int                 line_number_size;
    DVM_LineNumber      *line_number;
    int                 need_stack_size;
} DVM_Function;
```

type 是返回值的类型，name 是函数名。parameter 和 local_variable 用下面给出的结构体保存名称和类型。类型是在初始化局部变量时使用的，但是现在还没有用到。

```
typedef struct {
    char                *name;
    DVM_TypeSpecifier   *type;
} DVM_LocalVariable;
```

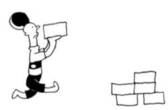
在 DVM_Function 中的 is_implemented 是一个标志。它表示“这个函数是否在这个 DVM_Executable 中实现”。

比如 print() 函数由原生函数组成，使用者编写的 Diksam 程序中并没有对其进行过定义，因此，这个函数对应的 DVM_Function 的 is_implemented 为 false。即使是这样的函数也要被登记到 DVM_Function 的对应表中，因为在函数调用的时候必须要通过 DVM_Function 对应表中的索引值。

DVM_Function 结构体的成员，指针 code 指向该函数对应的字节码。

6.3.9 顶层结构的字节码

DVM_Executable 结构体的成员，指针 code 指向顶层结构对应的字节码。



关于如何生成字节码将在 6.3.12 节中作介绍。

6.3.10 行号对应表

对于执行字节码的语言来说，发生错误时，如果不能提示发生的错误在源文件中的行号，对于使用者来说就太不友好了。因此，DVM_Executable 的成员 line_number 保存了字节码上的位置对应的源文件的行号。

这种对应关系的类型用 DVM_LineNumber 结构体表示，定义如下。

```
typedef struct {
    int line_number;/* 源代码的行号 */
    int start_pc;/* 字节码的开始位置 */
    int pc_count;/* 从 start_pc 开始，接下来有几个字节的指令对应着同一 line_number */
} DVM_LineNumber;
```

上述信息，在 generate.c 的 generate_code() 函数中与字节码同时生成。因为 1 行源代码通常会生成多个指令，所以在为同一行源代码生成编码时，不增加 DVM_LineNumber 对应表的元素，只增加 pc_count。

这里只保存了顶层结构的行号对应表，函数内的行号保存在各自的 DVM_Function 对应表中。

6.3.11 栈的需要量

在 crowbar 中，在每次进行入栈操作时，都会检查栈的空间。只要空间不足就会用 realloc() 进行扩展。

但是，Diksam 是执行字节码的语言，因此我们对它的执行速度还是有所期待的，所以我们在这里要避免“每次都进行栈空间检查”的做法。

因此，在 DVM_Executable 的成员 need_stack_size 中保存了顶层结构所需的栈空间大小。各函数需要的栈空间大小保存在 DVM_Function 对应表中。

这里的重点是：不论是顶层结构还是函数，它们所需要的栈空间大小都是在编译时决定的。

虽然在前面已经提到过，但是在这里还要再说一下，DVM 将完全信任编译器生成的字节码。另外，Diksam 的编译器绝对不会生成下例这样的字节码。

```
10 push_int_1byte 5
12 jump 10
```

在这个例子中，会无限循环地将 5 入栈，直到内存溢出。但是，Diksam 从语法上就杜绝了编写这种（能生成类似于上述例子中字节码的）源代码的可能。因此，扫描全部 push 系列的指令，并计算出 push 所需内存总量，用此方法就可以计算出顶层结构或者各函数所需的栈空间的大小了（在现在的实现中，并没有把出栈的内存计算在内，因此这个值会略大一些，但是多一些的空间并不会造成问题，所以就保持现状了）。各个指令消耗的栈空间量将会保存在 dvm_opcode_info 数组中（请参考 6.3.12 节）。

基于上述做法，检查栈大小有以下两个最佳时机。

- 程序开始执行时，检查栈空间是否能满足顶层结构的需要。
- 函数开始执行时，检查栈空间是否能满足这个函数的需要。在出现深层递归需要消耗大量栈空间的情况下，会数度进行检查，对栈空间的需求也会迅速地增长。

6.3.12 生成字节码（generate.c）

对于生成字节码来说，大部分麻烦事已经在 fix_tree.c 中做完了，因此 generate.c 要做的事情，大概就只剩下“按照自上而下的顺序遍历分析树然后吐出字节码”了。

下面我们就来看一下到底要“吐出”的是什么样的字节码。

■ 字节码的结构

在这个小节我将整理出至此为止一直没有明确的字节码的结构。

字节码由**命令**（指令，instruction）和**操作数**（operand）组成。

操作数可以想成是 C 等语言中函数的参数。比如在例子 push_int 10 中 push_int 指令处理了一个操作数，这个指令的操作数是常量池中的索引值，这个值是 10。

Diksam 的字节码以字节为单位（否则就不能叫“字节码”了）。指令也用一个字节来表示。

还是以上面的例子来说，push_int 对应的值是 3。这里用枚举类型 DVM_Opcode 来表示（DVM_code.h）。

```
typedef enum {
    DVM_PUSH_INT_1BYTE = 1,
    DVM_PUSH_INT_2BYTE,
    DVM_PUSH_INT,          ←这个是 push_int（也就是 3）
    DVM_PUSH_DOUBLE_0,
    DVM_PUSH_DOUBLE_1,
    DVM_PUSH_DOUBLE,
    DVM_PUSH_STRING
    （中间省略）
} DVM_Opcode;
```

操作数有以下三种。

- 1 个字节的整数。直接保存跟在指令后面的操作数。
- 两个字节的整数。把跟在指令后面的操作数作为大尾序保存。
- 常量池的索引值。该值现在是两个字节，把跟在指令后面的索引值作为大尾序保存。

什么样的命令处理什么样的操作数都定义在了 /share/opcode.c 中。

```
OpcodeInfo dvm_opcode_info[] = {
    {"dummy", "", 0},
    {"push_int_1byte", "b", 1},
    {"push_int_2byte", "s", 1},
    {"push_int", "p", 1},
    {"push_double_0", "", 1},
    {"push_double_1", "", 1},
    {"push_double", "p", 1},
    {"push_string", "p", 1},
    （以下省略）
```

这个数组用于调试时反编译功能（disassemble.c）之外的必须顺序分析字节码的情况。

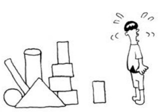
这里的 b 代表 1 个字节的整数，s 代表两个字节的整数，p 代表常量池的索引值。在表示字符串的时候，必须要使用取得多个操作数的指令。接下来的 0 和 1 的数值就是它们指令本身所需的栈空间。请参考 6.3.11 节。

■ 生成字节码

为了生成字节码，在 generate.c 中包含了以下函数。

```
static void
generate_code(OpcodeBuf *ob, int line_number, DVM_Opcode code, ...)
```

这个函数在获取指令和行号的同时，采用可变长参数来传递操作数。使用的例子如下。



```
/* 生成 push_int 的代码。
* cp_idx 是常量池的索引值。
*/
generate_code(ob, expr->line_number, DVM_PUSH_INT, cp_idx);
```

6.3.13 生成实际的编码

本节将要介绍 Diksam 源代码的组成元素实际会转换成什么样的字节码。

■ 标识符

标识符有以下几种。

- 局部变量
- 全局变量
- 函数

局部变量（是左值的情况下）将会生成如表 6-3 的字节码。操作数全部用两个字节的整数、栈上的索引值（基于 base 的偏移量）表示。

表 6-3 局部变量相关的指令

指令	含义
push_stack_int	将 int 类型的局部变量值保存到栈中
push_stack_double	将 double 类型的局部变量值保存到栈中
push_stack_string	将 string 类型的局部变量值保存到栈中

这三个指令只是针对不同的类型，但做的事情都一样。在枚举类型 DVM_Opcode 中，指令以 int、double、string 的顺序排列，因此生成字节码的操作可以用如下的方式进行。

```
/* push_stack_int */
generate_code(ob, expr->line_number,
              DVM_PUSH_STACK_INT
              + get_opcode_type_offset(expr->u.identifier
                                       .u.declaration
                                       ->type->basic_type),
              expr->u.identifier.u.declaration->variable_index);
```

get_opcode_type_offset() 函数，如果参数是 boolean 或者 int 则返回 0。如果是 double 则返回 1，string 则返回 2*。

当参数是 boolean 的时候也返回 0，这是因为虽然在 Diksam 语言中有 boolean 类型，但是 DVM 中并没有 boolean 类型，所以用 int 代替。

* 实际上在现在的实现中，局部变量保存在栈中，栈的实体是 DVM_Value 联合体的数组，因此这种指令本身没有必要根据类型区别使用。但是，考虑到要以字节为单位计算局部变量的地址，为了在实现上节省内存空间，在 Diksam 中还是分开变成了多个指令。

在全局变量中使用 push_（类型名）_static 指令代替 push_（类型名）_stack 指令。操作数是全局变量的索引值。

在函数的情况下，根据 push_function 的索引（DVM_Function 数组的下标）入栈，但是在执行时被改写成了别的方式。详细请参考 6.4.1 节。

■ 双目运算符

双目运算符生成的指令如表 6-4 所示。表中（类型）的部分，请看作是 int、double 或者 string，但是 string 只有相加和比较的运算。

表 6-4 双目运算符的指令

运算符	指令	含义
+	add_（类型）	加法运算
-	sub_（类型）	减法运算
*	mul_（类型）	乘法运算
/	div_（类型）	除法运算
%	mod_（类型）	模运算
==	eq_（类型）	等值比较
!=	ne_（类型）	不等值比较
<	gt_（类型）	小于
<=	ge_（类型）	小于等于
>	lt_（类型）	大于
>=	le_（类型）	大于等于
&&	logical_and	逻辑 AND
||	logical_or	逻辑 OR

这些指令没有操作数。

■ 单目运算符

单目运算符有单目的减号和逻辑非（!）。也可以把类型转换看作是一种单目运算符（目前会由编译器自动插入）。

单目运算符的指令如表 6-5 所示。

表 6-5 单目运算符的指令

指令	含义
minus_（类型）	符号反转
logical_not	逻辑 NOT
cast_int_to_double	将 int 转换为 double
cast_double_to_int	将 double 转换为 int
cast_boolean_to_string	将 boolean 转换为 string
cast_int_to_string	将 int 转换为 string
cast_double_to_string	将 double 转换为 string

■ 赋值

现阶段的 Diksam 中还没有数组和对象的成员，因此赋值必然是以“**变量名 = 表达式 ;**”的形式。

所以，我们首先计算右边，其结果值可以从栈中取得后，使用 `pop_stack_xxx` 或者 `pop_static_xxx` 让变量出栈。

与 C 语言相同，Diksam 的赋值本身也是在获取表达式的值（因此也有可能出现像 `a = b = c;` 这样的赋值方式）。也就是说，在赋值结束后，栈上肯定会留下一个值。计算右边的值，在出栈赋给变量后栈上就没有值了，因此需要使用 `duplicate` 指令复制栈顶的值并将其入栈。

但是，实际上很少有人会写 `a = b = c;` 这样的赋值语句（也可以说是因人而异吧），大部分的情况没有必要特意使用 `duplicate` 复制栈上的值。

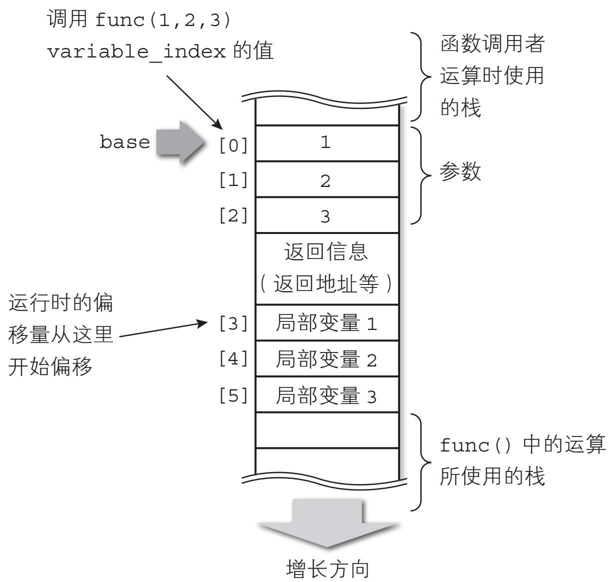
因此，生成赋值表达式的时候，要把“这个表达式是否是表达式语句的顶级表达式”作为标识进行传递，如果（当前表达式）是表达式语句的顶级就不需要使用 `duplicate` 了。

■ 函数调用

再看一下图 6-5 中的例子，Diksam 中函数调用时，栈会像图 6-6 所示进行增长。

首先，按照从前向后的顺序对参数进行计算，并将其入栈（Diksam 中没有可变长参数，因此没有必要像 C 语言一样从后面开始入栈）。

图 6-6
函数调用时的栈



接着，将“函数”入栈（这个“函数”其实是 DVM_VirtualMachine 结构体中 Function 的索引值。详细请参考 6.4.1 节）。正如前面写到的，Diksam 中调用函数使用 () 运算符，函数名本身就是一个表达式。使用 push_function 将这个表达式的值入栈。

然后，执行 invoke，从而执行被调用的函数。invoke 将保存着 push_function 的值从栈中移出，创建承载着返回信息的局部变量的内存空间，并让开始函数执行其一系列动作。

详细请参考 6.4.3 节 。

■ 控制结构

如 6.2.4 节中所述，像 if 语句这样的控制结构在字节码中是使用跳转命令来实现的。

为了实现跳转，就必须要知道跳转目标的地址。这里说的“地址”是指在 DVM_Executable 中的每个函数，或者是保存在顶层结构中的字节码的数组（DVM_Byte *code）的下标。

比如有下面这样一段代码。

```
int a;
if (a == 0) {
    a = 1;
} else {
    a = 5;
}
```

编译后生成如下字节码。

```
0  push_static_int   0   # 将变量 a 的值入栈
3  push_int_1byte    0   # 将 0 入栈
5  eq_int                # 比较
6  jump_if_false     17  # 如果不相等则跳转到 17
9  push_int_1byte    1   # 为了赋值将 1 入栈
11 pop_static_int    0   # 将 1 出栈赋值给 a
14 jump  22              # 跳到 22（这段代码的末尾）
17 push_int_1byte    5   # 为了赋值将 5 入栈
19 pop_static_int    0   # 将 5 出栈赋值给 a
```

这段代码中，最左边的数字就是“地址”。为了迎合机器语言中的说法，某些特定的地址使用“地址码 XX”的说法表示。

在上面的例子中，a 和 0 比较后，在地址码 6 的地方判断如果相等就跳转到地

址码 17。但是问题在于，在地址码 6 的 `jump_if_false` 命令生成的时候，还不知道要跳转的目标就是“地址码 17”。

因此，Diksam 的编译器采用了以下的方法。

1. 跳转命令等在必须使用地址的时候，使用`get_label()`函数取得一个“标签”。这里取得的“标签”是指，标签对应表的下标。
2. 字节码生成的最初阶段，在要写入跳转目标地址的地方写入暂定的标签。
3. 确定下来标签要代替的位置的地址后，使用`set_label()`函数，将地址存入标签对应表。
4. 字节码全部生成后，根据标签对应表，将写入暂定的标签的地方替换成真正的地址。

6.4 Diksam 虚拟机

编译（生成字节码）完成以后，就要放到 Diksam 虚拟机（DVM：Diksam Virtual Machine）中执行了。

DVM 在实现上使用`DVM_VirtualMachine` 结构体来表示（dvm_pri.h）。

```
struct DVM_VirtualMachine_tag {
    Stack        stack;
    Heap         heap;
    Static       static_v;
    int          pc;
    Function     *function;
    int          function_count;
    DVM_Executable  *executable;
};
```

结构体的前三个成员保存了 DVM 运行时的记忆空间。如代码所示，DVM 具有以下三个记忆空间。

1. 栈

前面已经说过，需要在栈上创建空间的有局部变量、函数的参数或函数返回的返回信息等。

`DVM_VirtualMachine` 结构体的成员 `stack`，它的类型 `Stack` 如下所示。

```
typedef struct {
    int          alloc_size;
```

```
    int         stack_pointer;
    DVM_Value   *stack;
    DVM_Boolean *pointer_flags;
} Stack;
```

一目了然，栈的实体就是 DVM_Value 类型的数组。

DVM_Value 相当于 crowbar 中的 CRB_Value，是一个能够保存所有（在 Diksam 中可以使用的）类型的联合体（DVM.h）。

```
typedef union {
    int         int_value;
    double      double_value;
    DVM_Object  *object;
} DVM_Value;
```

DVM_Value 与 CRB_Value 不同，不会根据类型打上不同标记。静态语言中类型是可以被识别的，因此不再需要标记类型了。

但是，在垃圾回收的时候，仅依靠静态的信息判断栈上的值是否是指针还是有些困难的，但也可以通过函数定义取得局部变量或者参数的类型。同样是保存在栈上，计算过程中的值的类型却很难弄清楚。

因此，Stack 结构体的 pointer_flags 数组保存着栈上的值是否是指针。pointer_flags 数组和 stack 的大小相同，可以使用同一个下标进行引用。

Stack 结构体的成员 stack_pointer 是**栈指示器**（stack pointer），它保存着栈顶的索引值。

栈指示器所指的是下次要入栈的元素的索引值。实际上已入栈元素的索引值是一个小于栈指示器 -1 的值。

2. 堆

堆是一个通过引用进行访问的内存区域。现在的 Diksam 中并不存在类和对象，因此现在的堆中只保存字符串。

和 crowbar 一样，堆上的对象以链表形式保存。

```
/* 保存对象的结构体 */
struct DVM_Object_tag {
    ObjectType  type;
    unsigned int        marked:1;
    union {
        DVM_String      string;
    } u;
    struct DVM_Object_tag *prev;
```

```
    struct DVM_Object_tag *next;
};

/* 堆的结构体 */
typedef struct {
    int         current_heap_size;
    int         current_threshold;
    DVM_Object  *header;
} Heap;
```

3. 静态（static）空间

DVM_VirtualMachine 的 static_v 成员用来保存全局变量（因为 static 和 C 语言的关键字冲突，所以成员的名字用 static_v 表示）。

Static 结构体的定义如下。

```
typedef struct {
    int         variable_count;
    DVM_Value   *variable;
} Static;
```

如代码所示，结构体中保存着 DVM_Value 的数组。

在使用 push_static_int 等指令引用全局变量时，作为操作数传递给指令的就是这个数组的下标。

另外，在 DVM_VirtualMachine 内用于嵌入到字节码中的值只能来自同一个数组的下标，也就是说，在现在的 Diksam 语言中，一个 DVM 只能对应一个 DVM_Executable。

但是很明显，在 DVM_VirtualMachine 内不只保存了一个 DVM_Executable。为了解决这个问题，有必要在多个源文件链接并执行的时候进行纠正（请参考 8.1.5 节）。

DVM_VirtualMachine 还有一个属性 pc 用来表示**程序计数器**（program counter）。

程序计数器在字节码中起到保存当前正在执行的指令地址的作用（这里说的“地址”是指保存着字节码的数组的下标）。

因此，在使用跳转命令时，只需要把程序计数器改写为要跳转的目标就可以了。但是，实际上 DVM_VirtualMachine 结构体的成员 pc 在程序开始执行后立刻就被复制到了局部变量，但在之后却并没有被回写回来，所以现在这个成员派不上什么用场。

其余的两个成员 function 和 executable 将在后面的章节中进行介绍。

6.4.1 加载 / 链接 DVM_Executable 到 DVM

在程序执行前，首先必须要为 DVM_VirtualMachine 绑定 DVM_Executable，用方法 DVM_add_executable() 来完成这项工作。由于一个 DVM_VirtualMachine 只能对应一个 DVM_Executable，因此这个函数的名字有点挂羊头卖狗肉的意思。

在 DVM_add_executable() 中会进行以下几个处理。

1. 将函数添加到 DVM_VirtualMachine 中

在这里我要重申一下，一个 DVM_VirtualMachine 只对应一个 DVM_Executable。但是，像 print() 这样的原生函数储存在了别的地方（即 DVM_Executable 以外的地方），因此有必要将函数以某种方式进行链接。

DVM_VirtualMachine 结构体的 function 数组正是为此而存在的对应表。

其类型 Function 的定义如下所示。

```
/* 将 Function 进行分类的标签 */
typedef enum {
    NATIVE_FUNCTION,
    DIKSAM_FUNCTION
} FunctionKind;

/* 保存 print() 之类的原生函数 */
typedef struct {
    DVM_NativeFunctionProc *proc;
    int arg_count;
} NativeFunction;

/* 引用在 Diksam 中定义的函数 */
typedef struct {
    DVM_Executable   *executable;
    int              index;/* 上层 executable 内（译注：与本函数对应的）DVM_
                              Function 的下标 */
} DiksamFunction;

/* Function 结构体本身 */
typedef struct {
```

```
    char            *name;
    FunctionKind    kind;
    union {
        NativeFunction native_f;
        DiksamFunction diksam_f;
    } u;
} Function;
```

DVM_Executable 中保存的函数在执行时使用对应表进行引用。

这个引用表中同样登记着像 print() 这样的原生函数，因此利用这个对应表的索引，不论是 Diksam 中定义的函数还是原生函数，全部可以引用到。

在字节码中调用函数的时候，使用 push_function 指令将函数对应的索引值入栈，这个被嵌入的值，就是在编译时 DVM_Executable 内 DVM_Function 数组的下标。

DVM_Function 数组中不包含当前源代码中使用的原生函数，因此 Function 数组和索引值就会出现差异。下一步就是要修正这个问题。

2. 替换函数的索引

如前面所述，在编译阶段函数的索引对于当前 DVM_Executable 来说是局部的，在执行时会被汇总到一个数组中，因此必须要将索引进行转换。

这个操作会直接替换字节码中的操作数。

但是，如果直接调整成与某个 DVM_VirtualMachine 匹配的字节码的话，当一个 DVM_Executable 对应多个 DVM_VirtualMachine 时就会出现问题。这个问题将在 9.3.4 节中修正。

3. 修正局部变量的索引值

在进行上述操作的同时，也会修正局部变量的索引值。

如图 6-5 所示，引用局部变量时，索引与参数之间隔着返回信息。但是，编译器并不知道返回信息的大小（其实是可以知道的，但是为了信息保密而使编译器不能获取到返回信息的大小），因此在编译时，参数的索引会接着引用后面的索引值。

DVM_add_executable() 中会使用 push_stack_xxx 和 pop_stack_xxx 指令对此进行修正。

4. 将全局变量添加到 DVM_VirtualMachine 中

只是用 DVM_Executable 结构体的 global_variable 数组的个数创

建 DVM_VirtualMachine 结构体的 static_v.variable 数组，并初始化数组的值。

6.4.2 执行——巨大的 switch case

接下来就要开始执行了。

DVM 是一个将编译器生成的字节码逐个执行的虚拟机。也就是说，只要循环地执行一个与字节码的指令种类一样多的巨大 switch case 就可以了。

关于这个话题，可能直接看代码会更形象（代码清单 6-2）。

代码清单 6-2
execute()

```
 static DVM_Value
execute(DVM_VirtualMachine *dvm, Function *func,
        DVM_Byte *code, int code_size)
{
    int             pc;
    int             base;
    DVM_Value       ret;
    DVM_Value       *stack;
    DVM_Executable  *exe;

    stack = dvm->stack.stack;
    exe = dvm->executable;

    for (pc = dvm->pc; pc < code_size; ) {
        switch (code[pc]) {
        case DVM_PUSH_INT_1BYTE:
            STI_WRITE(dvm, 0, code[pc+1]);
            dvm->stack.stack_pointer++;
            pc += 2;
            break;
        case DVM_PUSH_INT_2BYTE:
            STI_WRITE(dvm, 0, GET_2BYTE_INT(&code[pc+1]));
            dvm->stack.stack_pointer++;
            pc += 3;
            break;
        case DVM_PUSH_INT:
            STI_WRITE(dvm, 0,
                      exe->constant_pool[GET_2BYTE_INT(&code[pc+1])].u.c_int);
            dvm->stack.stack_pointer++;
            pc += 3;
            break;
        case DVM_PUSH_DOUBLE_0:
            STD_WRITE(dvm, 0, 0.0);
```

```
            dvm->stack.stack_pointer++;
            pc++;
            break;
        case DVM_PUSH_DOUBLE_1:
            STD_WRITE(dvm, 0, 1.0);
            dvm->stack.stack_pointer++;
            pc++;
            break;
        case DVM_PUSH_DOUBLE:
            STD_WRITE(dvm, 0,
                      exe->constant_pool[GET_2BYTE_INT(&code[pc+1])].u.c_double);
            dvm->stack.stack_pointer++;
            pc += 3;
            break;
        case DVM_PUSH_STRING:
            STO_WRITE(dvm, 0,
                      dvm_literal_to_dvm_string_i(dvm,
                                                  exe->constant_pool
                                                  [GET_2BYTE_INT(&code[pc+1])]
                                                  .u.c_string));
            dvm->stack.stack_pointer++;
            pc += 3;
            break;
    （之后省略）
```

现在这个函数（execute.c 的 execute() 函数）就已经有 400 多行了，并且还会不断增加。如果有“一个函数必须少于 XX 行”等这样机械的编码规约的话，那么在实际工程中程序员们肯定会违反这个规则。

但我觉得把一个函数分割开并没有让它变得更易读。即使是规定了“一个函数必须少于 XX 行”这样机械的编码规约，也不能说编程本身是一项机械的工作。

更重要的是，由于这个函数的内部引用了栈，因此用到了以下这些宏。

- STI(dvm, sp), STD(dvm, sp), STO(dvm, sp)
  以当前栈指针加上 sp 为索引值，返回栈上对应元素值。主要用于四则运算等，也用于双目 / 单目运算符操作栈顶附近的值。
  STI 用于 int，STD 用于 double，STO 用于 string（对象）。下述三个方法同理。

- STI_I(dvm, sp), STD_I(dvm, sp), STO_I(dvm, sp)
  直接以 sp 为索引值，取得栈上对应元素值。使用了 push_stack_xxx、pop_stack_xxx 系列的指令。
  这些指令不是用来引用栈顶附近的值的，而是用来引用以 base 为起点的索引值

对应的栈元素的。

- STI_WRITE(dvm, sp, r)，STD_WRITE(dvm, sp, r)，STO_WRITE(dvm, sp, r)
  用与 STI() 等相同的方法来指定栈上的元素，并在对应元素的位置写入 r。因为使用了 STI() 等宏命令，所以可以用 STI(dvm, 0) == xxx 的形式进行赋值。但是，由于必须要根据类型是否为指针来设定栈的 pointer_flag，因此特意制作了用来写入的宏。

- STI_WRITE_I(dvm, sp, r)，STD_WRITE_I(dvm, sp, r)，STO_WRITE_I(dvm, sp, r)
  直接以 sp 为索引值的 STx_WRITE()。

6.4.3 函数调用

作为程序的起始点，execute() 函数确实是一个巨大的函数，逐个地执行每个指令绝对不是几行代码就能解决的。

这里将要说明的是一个稍微复杂一些的话题——函数调用。

函数调用按照以下的顺序执行。

1. 将参数以从前向后的顺序入栈。
2. 使用push_function将函数的索引值入栈。

上述操作执行后，栈的状态如图 6-7 所示。

然后执行 invoke 指令。

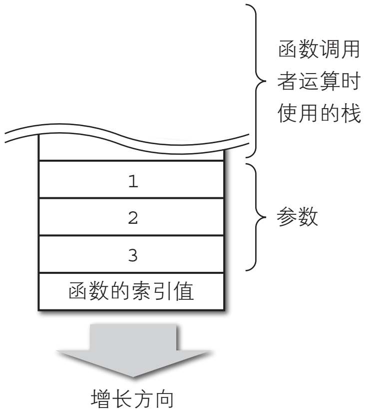
3. 执行invoke，调用栈顶的函数。

如果被调用的函数是一个原生函数，那么上述操作就会实际地执行这个原生函数。

Diksam 原生函数的调用形式如下（以 print() 为例）。

```
static DVM_Value
nv_print_proc(DVM_VirtualMachine *dvm,
              int arg_count, DVM_Value *args)
{
```

图 6-7
函数调用（1）



这里把 DVM_Value 的数组作为参数传递给了 nv_print_proc 函数，但其实，参数在栈上正好是按顺序排列的，因此这里也可以只传第一个参数的指针。

如果要调用 Diksam 的函数，要进行以下操作。

4. 将返回信息入栈。
5. 设置base的值。
6. 初始化局部变量。
7. 替换执行中的executable和函数。
8. 将程序计数器置为0并开始执行。

使用 CallInfo 结构体来表示之前一直在说的返回信息的实体。

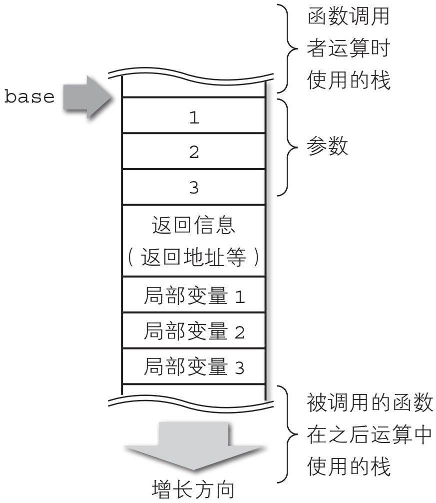
```
typedef struct {
    Function    *caller;
    int         caller_address;
    int         base;
} CallInfo;
```

caller 指向当前函数的调用者（也是函数），caller_address 指向函数内字节码上的地址。base 指向调用者的 base 值（引用参数或者局部变量的起点）。

在函数被调用的时候，因为栈中还保存着很多运算过程中的值，所以如果 CallInfo 中不保存 base 的话，函数结束后就无法返回了（因为不知道返回到哪里）。

对于 CallInfo 结构体来说，首先要被覆盖设置调用函数的索引（由 push_function 入栈的）。之后设置新 base，创建局部变量的内存区域。在被调用的函数开始执行时，栈的状态如图 6-8 所示。

图 6-8
函数调用（2）



被调用的函数在达到这个状态后就可以开始执行了。

反过来，从函数中 `return` 的时候是怎样操作的呢？函数在最后结束之前要先执行 `return`，因此函数在结束时必须在局部变量的下一个位置中保存返回值（如图 6-9）。

将参数、`CallInfo`、局部变量全部移除后，将返回值移动到栈顶。这样做，即使函数是在表达式的计算过程中被调用的，也可以让它正确地使用函数的返回值（如图 6-10）。

图 6-9
函数调用（3）

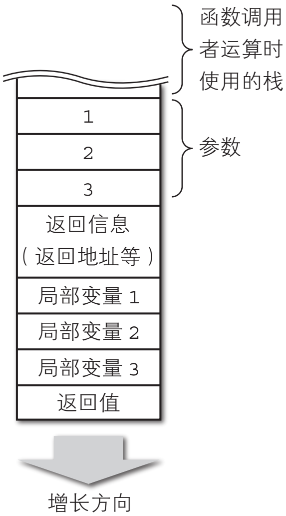
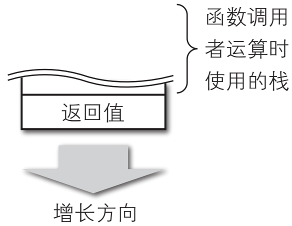
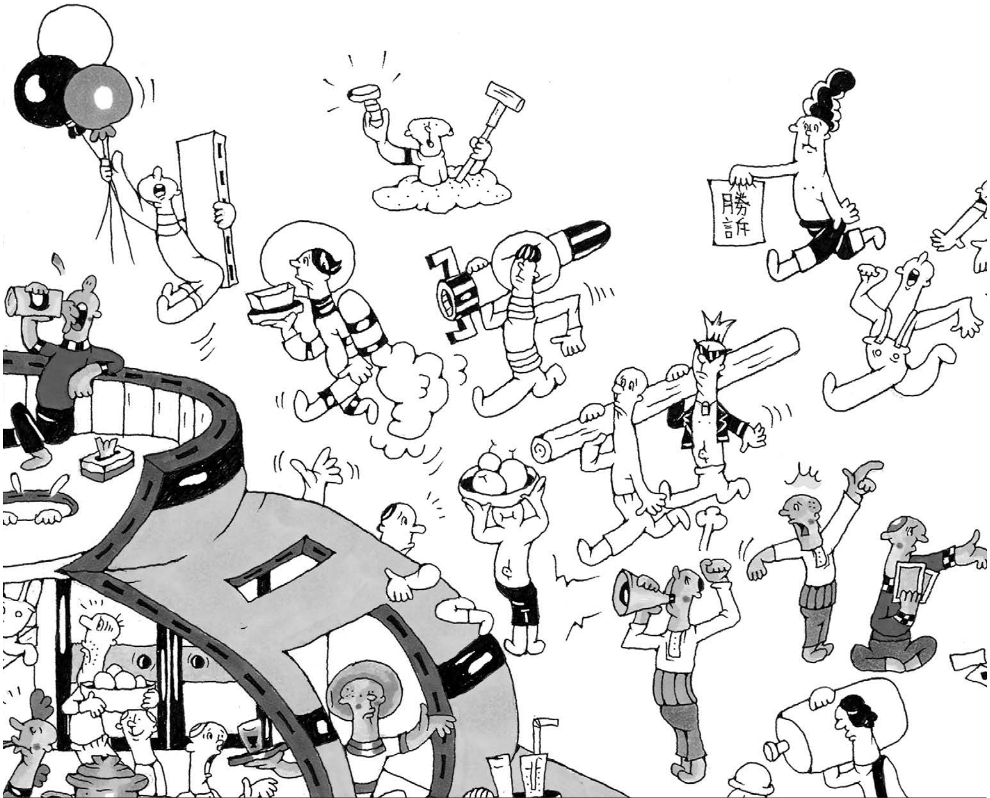


图 6-10
函数调用（4）





第 7 章

为 Diksam 引入数组

7.1 Diksam 中数组的设计

由于 Diksam book_ver.0.1 不能使用数组，因此让人感觉不太实用，所以在 book_ver.0.2 中我们将引入数组的概念。啊，这个开场白好像和 4.1 节的一样呢。

7.1.1 声明数组类型的变量

Diksam 中数组的设计与 Java 大致相同。

首先，在 Diksam 中变量必须要进行声明，当然数组类型的变量也不例外，需要用 Java 的风格进行声明。

```
int[] a;// 声明 int 类型的数组
```

创建数组时的语法也和 Java 一样。

```
// 创建了一个可以访问到 a[2][4] 的数组
a = new  int[3][5];
```

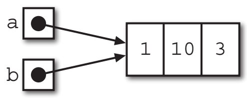
与 crowbar 和 Java 一样，Diksam 的数组也是引用类型。因此，下面的代码会输出 a[1]..10。

```
int[] a = {1, 2, 3};
int[] b = a;// a 和 b 指向同一个数组

// 因此，改变 b[1] 的话 a[1] 也会跟着改变
b[1] = 10;
print("a[1].." + a[1] + "\n");
```

因为数组 a 和数组 b 指向了同一个数组，所以输出这样的结果也是理所当然的（如图 7-1）。

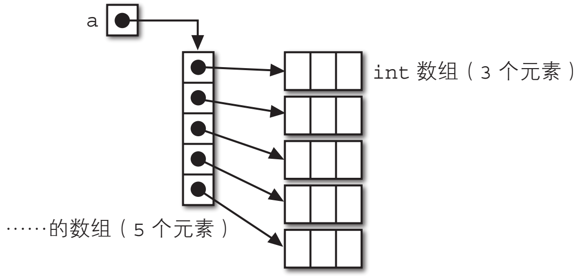
图 7-1
两个变量同时引用一个 Diksam 的数组



另外，（看上去是）多维数组实际上是数组的数组。

总之，在 a = new int[3][5]; 这段代码中，a 最后得到的是“int 数组（3 个元素）的数组（5 个元素）”。关于数组元素的引用形式，如图 7-2 所示。

图 7-2
Diksam 的多维数组



7.1.2 数组字面量

在 Java 中，数组类型变量只有在声明的同时进行初始化，才可以用如下方式。

```
int[] a = {1, 2, 3};
```

其他情况下，数组字面量必须使用以下方式声明。

```
a = new int[]{1, 2, 3};
```

虽然知道在 Java（或 Diksam）这样的静态类型语言中必须明确地指定类型，但是总觉得 new int[] 这部分太冗长了，另外对于初学者来说，初始化和其他情况不同也容易造成混乱。更重要的是我（从语言实现者的角度出发）不支持一种常量有两种声明方式。

因此在 Diksam 中，数组字面量的类型由“最初的元素的类型”决定（这里模仿的是 D 语言）。

总之，下面这段代码中第 2 个和第 3 个元素会被转换成 double，以 double 型数组 {1.0, 2.0, 3.0} 的形式赋值给 a。

```
double[] a = {1.0, 2, 3}
```

第 2 和第 3 个元素会被转换为 double，被赋值给 a 的是一个由 {1.0, 2.0, 3.0} 组成的 double 数组。

在 Diksam 中（与 Java 相同），还没有决定元素值的数组，写作 a = new int[5][3];，这段代码创建了一个访问范围是从 a[0][0] 到 a[4][2] 的数组。

补充知识　D 语言的数组

D 语言是 Digital Mars 公司作为 C 语言的后继开发出来的编程语言。

在 D 语言中，如果想要创建一个访问范围从 `a[0][0]` 到 `a[4][2]` 的数组，就要写成 `int[3][5];`。而且，并不是堆中而是作为静态或者局部变量时数组的声明语法。如果要使用 new 进行动态分配时就要写成 `new int[3][5]` 了。

与 C 和 Java 一样，D 语言的数组下标也是从 0 开始，下标的上限和数组的大小相差 1。这点虽然很好，但是可以访问到 `a[4][2]` 的数组声明方式却是 `int[3][5];`，肯定有人会怀疑是不是把顺序搞错了。确实，在 C 语言中可以访问到 `a[4][2]` 的数组声明方式是 `int[5][3];`，Java 在 new 数组的时候也是 `int[5][3];`。

但是，在 Java（或者是 Diksam）中，`new int[5][3];` 得到的是 int 的数组（3 个元素）的数组（5 个元素）。Java（或者是 Diksam）的语法不可以从左边开始读，这点在 D 语言中正好相反。

虽然是这样，但是 Java 语言比 D 语言使用范围更广泛，这部分的语法 C# 和 Java 也是相同的，更重要的是，已经习惯了这样（Java）的写法突然改变的话，会变得混乱（我自己也会），因此在数组声明方式上，Diksam 是迎合了 Java 的做法。

虽然如此，D 语言是美国人开发出来的语言，在他们看来 D 语言这样的顺序可能更自然一点。

7.2 修改编译器

7.2.1 数组的语法规则

这次增加的语法规则如下所示。

1. 扩展类型标识符（`type_specifier`）以声明数组类型的变量
2. 使用new创建数组的语法（`array_creation`）
3. 数组字面量（`array_literal`）
4. 使用下标运算符（如`a[10]`）引用数组元素的语法

第 1 点，原来的类型标识符是下面这样的。

```
type_specifier
        : BOOLEAN_T
        | INT_T
```

```
                | DOUBLE_T
                | STRING_T
                ;
```

现在，像 boolean 或者 int 这样的基本类型都将被作为 basic_type_specifier，如下所示。

```
type_specifier
            : basic_type_specifier
            | type_specifier LB RB
            ;
```

LB 和 RB 是 Left Bracket 和 Right Bracket 的简称，代表 [和]。

type_specifier 可以包含 [] 本身。这样一来不论加几个 [] 都可以（例如 int[][][]）。

第 2 点，使用 new 创建数组的语法看上去好像挺麻烦的。

比如在 Java 中代码 new int[10] 会得到 int 数组（10 个元素）。

在此基础上如果加上下标 [5]，就变成了 new int[10][5]。这行代码会取得 new int[10] 的第 5 个元素 * 是不可能的，这当然是创建二维数组的意思。

* 因为数组的下标是从 0 开始的，所以这里用一般的计数方法取得的应该是第 6 个元素。

总之，下标运算符 [] 除了必须要适用于普通的表达式之外，还要适用基于 new 的数组创建（array_creation）语法。

在 Diksam book_ver.0.1 中组成表达式的最小元素是 primary_expression（运算符优先顺序最高的块），因此“基于 new 创建数组”被当作“例外情况”来处理。

```
primary_expression
        : primary_no_new_array /* 基于 new 创建数组之外的表达式 */
        | array_creation /* 使用 new 创建数组的表达式 */
        ;
```

引用数组元素的语法规则如下所示。下标运算符 [] 不被局限于使用 new 来创建数组。

```
primary_no_new_array
        /* 可以使用 [] 的只有 primary_no_new_array */
        : primary_no_new_array LB expression RB
       （之后省略）
```

……这是一段多么了不起的代码啊。可这不是我写的，我只不过是照搬了 Java 的语法规则而已 [7]。

7.2.2 TypeSpecifier 结构体

在 Diksam 的编译器中，使用 TypeSpecifier 结构体保存数据类型。

关于 TypeSpecifier 结构体请参考 6.3.4 节。

要点在于，要使用保存基础类型（DVM_BasicType）的 TypeSpecifier 结构体和链表连接起来的派生类型（TypeDerive）来表示所有数据类型。

这部分的代码如代码清单 7-1 所示。

代码清单 7-1
TypeSpecifier（Diksam book_ver.0.2 版）

```
typedef enum {
    FUNCTION_DERIVE,
    ARRAY_DERIVE
} DeriveTag;

typedef struct {
    ParameterList   *parameter_list;
} FunctionDerive;

typedef struct {
    int dummy; /* make compiler happy */
} ArrayDerive;

typedef struct TypeDerive_tag {
    DeriveTag   tag;
    union {
        FunctionDerive  function_d;
        ArrayDerive     array_d;
    } u;
    struct TypeDerive_tag   *next;
} TypeDerive;

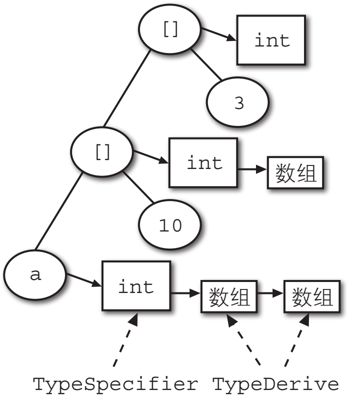
struct TypeSpecifier_tag {
    DVM_BasicType   basic_type;
    TypeDerive  *derive;
};
```

之前的派生类型只有“函数类型”，这次增加了数组的派生（在 TypeDerive 的 tag 中加入了 ARRAY_DERIVE）。

在 fix_tree.c 中，表达式的各个节点中也要附加对应地 TypeSpecifier 结构体。比如，使用 int[][]　a；声明变量 a，在附加 TypeSpecifier 的时候，首先给 basic_type 赋值为 DVM_INT_TYPE，在此基础上给进行了数组派生的 TypeDerive 累加上两个链表。

于是，使用下标运算符进行引用（如 a[10]）的时候，移除 TypeDerive 链表的第一个元素后剩下的就是表达式的类型了。同理，如果是 a[10][3] 的话，把两个都移除后，表达式的类型就是 int 了（如图 7-3）。

图 7-3
含有数组的分析树的类型



在图 7-3 中，圆形中间带有 [] 的符号表示下标运算符 IndexExpression。它是 Expression 结构体中的一种联合体，数组和下标的表达式保存在下面的结构体中。

```
typedef struct {
    Expression  *array; /* 数组的表达式 */
    Expression  *index; /* 下标的表达式 */
} IndexExpression;
```

7.3 修改 DVM

7.3.1 增加指令

由于这次引入了数组，因此在 DVM 中也要增加相应的指令，增加的指令如表 7-1* 所示。

* 请参考附录 C 中的范例阅读本表。

表 7-1 随着引入数组增加的指令

指令	操作数类型	含义	栈
push_array_int		根据栈顶的数组和下标，取得数组的元素（int 型）并将其入栈	[array int]→[int]
push_array_double		根据栈顶的数组和下标，取得数组的元素（double 型）并将其入栈	[array int]→[double]
push_array_object		根据栈顶的数组和下标，取得数组的元素（object 型）并将其入栈	[array int]→[object]
pop_array_int		将栈顶的值（int1）赋值给与数组（array）下标 int2 对应的元素	[int1 array int2]→[]
pop_array_double		将栈顶的值（double）赋值给与数组（array）下标 int 对应的元素	[double array int]→[]
pop_array_object		将栈顶的值（object）赋值给与数组（array）下标 int 对应的元素	[object array int]→[]
new_array	byte、short	创建以操作数 byte 指定维数,以 short 指定类型的数组,在栈中创建指定大小的空间并将数组入栈	[size1 size2 ...]→[array]
new_array_literal_int	short	使用已经入栈的操作数作为 int 型元素（操作数用来指定元素个数）创建数组并将数组入栈	[int1 int2 int3 ...]→[array]
new_array_literal_double	short	使用给定数量的已经入栈的操作数作为 double 型元素创建数组并将其入栈	[double1 double2 double3 ...]→[array]
new_array_literal_object	short	使用给定数量的已经入栈的操作数作为 object 型元素创建数组并将其入栈	[object1 object2 object3 ...]→[array]

在指令中出现了“object 型”的概念。它是在之前只包含了字符串的引用类型的基础上又增加了数组，是由字符串和数组组成的类型。

随着上述改变，除了专门处理字符串的操作（如字符串比较等），之前的 push_static_string 等指令都要重命名为 push_static_object 等了。

表 7-1 的指令中，我想必须要特别说明一下 new_array。

在表 7-1 中提到了“操作数 short 代表的类型”，这个操作数是指这次在 DVM_Executable 中新增的 DVM_TypeSpecifier 数组的下标（代码清单 7-2）。

代码清单 7-2
DVM_Executable
（book_ver.0.2）

```
struct DVM_Executable_tag {
    int                     constant_pool_count;
    DVM_ConstantPool        *constant_pool;
    int                     global_variable_count;
    DVM_Variable            *global_variable;
    int                     function_count;
    DVM_Function            *function;
    int                     type_specifier_count;    ← 新增
    DVM_TypeSpecifier       *type_specifier;         ← 新增
    int                     code_size;
    DVM_Byte                *code;
    int                     line_number_size;
    DVM_LineNumber          *line_number;
    int                     need_stack_size;
};
```

DVM_TypeSpecifier 结构体在 book_ver.0.1 时就已经存在了，它和 TypeSpecifier 结构体保存着同样的信息。

例如，使用 new int[5][3] 创建一个数组，new_array 的操作数将被指定为保存着 int[][] 类型信息的 DVM_TypeSpecifier 的下标。

这样一来，只要知道对应的 TypeSpecifier 就能够知道数组的维数，int[][] 型的数组也可以像 new int[5][] 这样，在代码运行过程中再创建另外一维。实际创建的维数（这里是 1）使用另外一个 byte 型操作数传递指令。另外，使用代码 a = new int[5][]; 创建的数组和 Java 一样，a[0]~a[4] 被初始化为 null。

补充知识　创建 Java 的数组字面量

如表 7-1 所示，在 DVM 中，new_array_literal_int 等创建常量的指令，会先将组成数组的值入栈，再利用已经在栈上的值创建数组。但是，JVM 就没有与此对应的指令。那么，Java 中是如何通过构造函数创建的数组或者使用 new int {1,2,3} 这样的代码创建数组的呢？让我们使用 javap 来看一下。

最初的代码：

```
class Test {
    public static void main(String[] args){
        int[] a = {1, 2, 3, 4, 5};
    }
}
```

javap的结果（只截取了指令部分）：

```
 0: iconst_5
 1: newarray int
 3: dup
 4: iconst_0
 5: iconst_1
 6: iastore
 7: dup
 8: iconst_1
 9: iconst_2
10: iastore
11: dup
12: iconst_2
13: iconst_3
14: iastore
15: dup
16: iconst_3
17: iconst_4
18: iastore
19: dup
20: iconst_4
21: iconst_5
22: iastore
23: astore_1
24: return
```

也就是说，相当于下面这段代码。

```
int[] a = new int[5];

a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 4;
a[4] = 5;
```

在我看来，生成字节码的体积太大了。在 Java 中，与一个方法对应的字节码是有大小限制的（Diksam 也一样），所以自动生成代码的时候（也许还有其他情况）可能会引起问题。

补充知识　C 语言中数组的初始化

Diksam 也好，Java 也好，数组字面量以及利用构造函数创建的数组，它们的内容都是在“运行时”决定的。所以，在下面这段代码中：

```
int[] a = {b * 10, func()};
```

从这段初始化的程序可以看出，数组元素可能只在运行时才能决定其值的表达式。

对此，在 C 中利用初始化程序初始化数组的时候，元素的内容必须是常量表达式。

因为有了这个限制，在编译时可以预先创建数组的内存映像，static 变量开始执行、自动变量[①]进入函数时，可以利用事先创建的内存映像进行初始化。

7.3.2 对象

在 book_ver.0.1 中可以称为对象的只有字符串，现在在 DVM_Object 中增加了数组成员，以对应这次新增的数组概念。

```
struct DVM_Object_tag {
    ObjectType  type;
    unsigned int    marked:1;
    union {
        DVM_String  string;
        DVM_Array   array;  ←新增
    } u;
    struct DVM_Object_tag *prev;
    struct DVM_Object_tag *next;
};
```

DVM_Array 的内容如下所示。

```
typedef enum {
    INT_ARRAY = 1,
    DOUBLE_ARRAY,
    OBJECT_ARRAY,
} ArrayType;

struct DVM_Array_tag {
    ArrayType   type;
    int         size;
    int         alloc_size;
    union {
```

① 一般情况下可以看作是局部变量。——译者注

```
        int         *int_array;
        double      *double_array;
        DVM_Object  **object;
    } u;
};
```

在 crowbar 中，数组是“CRB_Value 的数组[①]”。这次也一样，因为有 DVM_Value 联合体，数组也可以表现为数组。但是，由于 Diksam 是静态语言，因此数组的类型是静态决定的。绝对不可能把 double 加入到 int 的数组中。

这么说的话，int 的数组使用“sizeof(int) × 元素数”就可以毫无浪费地创建内存空间，即使是传递给 C 的内置例程处理起来也很舒适。因此，枚举类型 ArrayType 中的每个对象都表示不同数组元素的类型。ArrayType 没有必要对应 Diksam 中的所有类型。例如字符串的数组，或者是数组的数组，这些都是 OBJECT_ARRAY。数组的类型在编译时决定，因此运行时在这里没有必要保存严格的类型。现在的情况是，数组的类型信息只有 GC 用到了。

补充知识 ArrayStoreException

前面写到，在 Diksam 中既有字符串数组也有数组的数组，数组的对象中只保存了“OBJECT_ARRAY”这一个信息。与此相对，在 Java 中，数组对象中保存着完整的类型信息。这样的区别是基于以下两点原因。

- **Diksam 中还不存在类和继承，但是在 Java 中存在。**
- **在 Java 中，当 A 是 B 的子类时，A[] 也自动地成为了 B[] 的子类。**

例如有一个表示图形的类 Shape，有两个继承它的子类 Line 和 Circle。这时在 Java 中，可以把 Line 的数组赋值给 Shape[] 型的变量。这种设计乍看是挺方便的，实际上问题重重。请思考如下这个代码片段。

```
1:  Line[] lines = new Line[10];
2:  Shape[] shapes = lines;
3:  shapes[3] = new Circle();
4:  lines[3].startPoint = new Point(x, y);
```

第 1 行当然是合法的。第 2 行也一样，Line[] 是 Shape[] 的子类，因此在 Java 中也是合法的。第 3 行，因为 Circle 也是 Shape 的子类，所以 Java 在编译时并不会报错（更确切地说是报不出错）。

接下来的第 4 行就悲剧了。shapes 和 lines 指向同一个数组，因此 lines[3] 也就是 shapes[3]，它在第 3 行被赋了一个 Circle 对象的值。但是，在第 4 行的时候又要引用 Line 的起点（startPoint），Circle 中并没有 startPoint，因此

① 这个数组值必须是双向链表。——译者注

这行代码不能被执行。但是，编译器却始终认为 line[3] 肯定是 Line，因此编译时不会出现报错。 正因如此，在 Java 中，执行到第 3 行代码时会发生运行时的异常，ArrayStoreException。

只有在运行时掌握“这个数组在变量声明上是 Shape[]，但是它实际上却是 Line[]”，才能在实现时抛出上述异常。因此，在 Java 中，必须将完整的类型信息保存在数组的对象中。

我认为这是一个不良的设计。既然是静态语言，就应该在编译时完成类型检查，在运行时抛出异常不是很奇怪的吗？总之，我认为 Java 的“A 是 B 的子类时，A[] 也自动地成为了 B[] 的子类”这个规则是错误的。

Diksam 将在下一章引入类的概念，但是没有建立上述规则，因此也没有必要在数组中保存严格的类型信息。

7.3.3 增加 null

由于数组和字符串都是引用类型，因此增加了 null。

随之改变的是，以前字符串变量的初始值是空字符串，现在变成了 null。

关于 null 的规则如下所示。

1. 字符串类型、数组类型的变量可以赋值为null。
2. 字符串类型与值为null的变量用+连接的话，null会转换为字符串"null"。
3. 字符串类型与字面量null用+连接的话，null会转换为字符串"null"。
4. 字符串类型与数组类型可以和null进行比较。

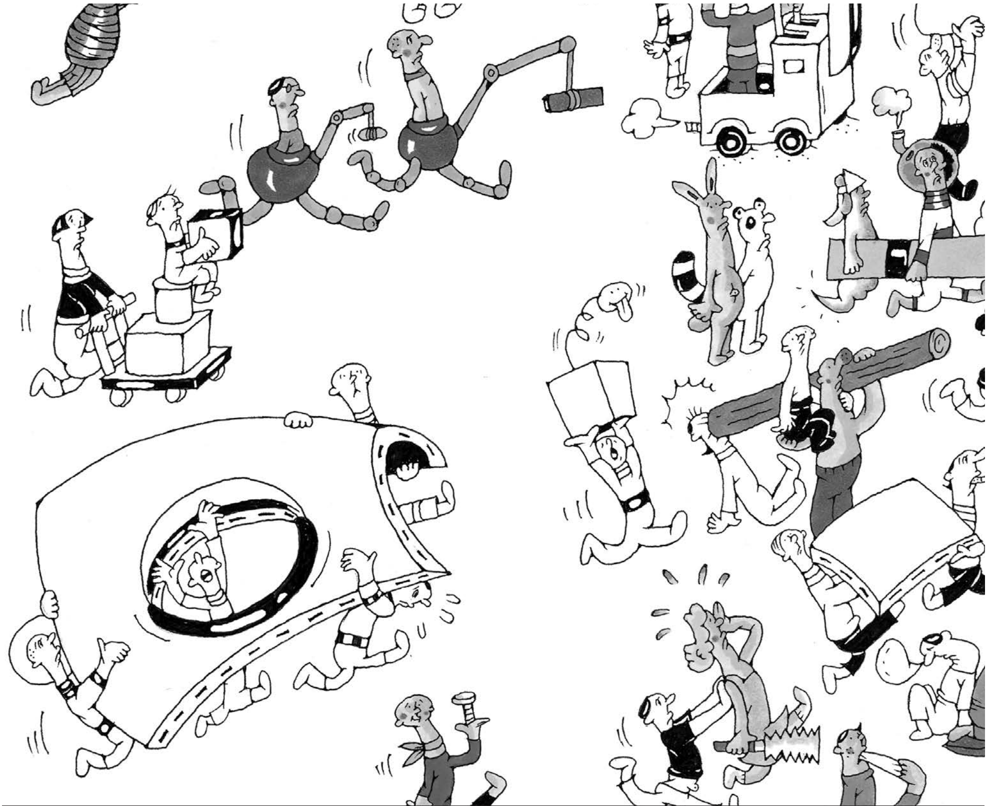
7.3.4 哎！还缺点什么吧？

这次引入了数组的概念，如果是了解当今编程语言的人肯定在期待着下面这些功能。既然引入了数组的概念，怎么能没有它们呢？

- 没有知道数组大小的（array.size() 或者 array.length 等）方法吗？
- 没有动态增加数组元素（例如 array.add(5)）的方法吗？
- 不能把数组内容直接输出（例如 print("array.." + array)）吗？

这些大概在当今的编程语言中都能实现（有些语言会把数组理所当然地输出为地址或者哈希值），说起来在 crowbar 中也实现了，但是这次却搁置起来了。这是因为考虑到这些功能最终都归结为“方法”，因此还是和类一起制作更为恰当。

所以，下一章将要对类进行处理。



第 8 章

将类引入 Diksam

8.1 分割源文件

本章的标题是“将类引入 Diksam”，在现在的 Diksam 中源代码不能分散地写在多个文件中。即使在编程语言中引入了类的概念，如果必须把所有代码都写在一个源文件中，那么这个语言又能有多大用处呢？

因此，首先要实现对源文件的分割。

8.1.1 包和分割源代码

分割源代码的方法，最简单的就是和 C 语言里面的 `#include` 一样，嵌入来自于其他源文件的代码。这个方法既简单又直接，非常实用。

但是，使用这个方法嵌入多个库文件时，函数名、变量名等很可能发生冲突。因此，在分割源代码的同时，加入相当于 Java 的包或者 C++ 和 C# 的命名空间的功能。

Diksam 的包的设计方式如下所示。

1. require

一些源文件如果需要其他源文件提供的功能时，应在该源文件的开头加入如下代码。

```
require hoge;
```

在这个例子中，编译器会在编译时搜索文件名为 hoge.dkh 的文件。和 Java 的 `import` 一样，`require` 也只能写在代码的开头。另外，`require` 读取的文件必须以 .dkh 为后缀，它是与 C 语言的头文件相似的文件。

搜索源文件的目录配置在环境变量 `DKM_REQUIRE_SEARCH_PATH` 中，多个搜索目录之间在 UNIX 中用冒号、在 Windows 中用分号分割。如果没有配置这个环境变量的话，将在当前目录（.）中进行搜索。这里的设计方式基本上和 Java 的 `CLASSPATH` 相同。

被 `require` 的文件有可能还要 `require` 其他文件，这时不会对同一个文件进行重复读取。

2. 动态加载

对于 require 的文件来说，虽然也可以把必要的函数的源代码全都写在里面，但是函数只要像下面这样进行**签名声明**也可以编译通过。

```
int print(string str);
```

如果是像 print() 这样的原生函数，签名声明后就可以直接使用了。

如果不是原生函数的话，只有在函数**被调用的时候**才会加载对应的源代码。这种方式称为**动态加载**(dynamic load)。因为程序中总有些功能是不常用的，使用了动态加载后相信能够实现高速化启动。

如果在 hoge.dkh 中进行了签名声明，那么在函数被调用的时候会对 hoge.dkm 进行搜索。在创建库文件时，.dkm 实现了 .dkh 中定义的设计。

动态加载时搜索的目录并不配置在环境变量 DKM_REQUIRE_SEARCH_PATH 中，而是从 DKM_LOAD_SEARCH_PATH 中获取。在这里，特意使用两个不同的物理路径来区分库文件的设计和实现。另外，也可以使用“在测试过程中将实现文件作为存根”的方法。

实际上，.dkh 文件作为设计公开的大小与其实现（ .dkm 文件）后的大小相差悬殊，.dkh 和 .dkm 的对应关系应该是 1：n 的样子，但是这样一来，在动态加载时就需要另外指定搜索源代码的方法了，因此这里先让它们保持 1：1 的状态。我认为动态加载是能够使用 Diksam 编写大程序的一个先决条件。

3. 包

在 Diksam 中，一个源文件就对应着一个包（在 .dkh 和 .dkm 分开编写的情况下，它们两个的代码要在一个包中）。

像 Java 那样在每个源文件的开头都要逐个对包进行声明是非常麻烦的，通常情况下，Java 的目录层级和包的层级一致，也就是把同样的信息体现在了两个地方。这样一来，在修改时就会出现问题（尤其是 Java 的包名，使用起来像是互联网的域名）。既然如此，单纯地使用源文件名和包名的组合可能更简单。

Diksam 的包名使用点（ . ）进行分割，根据包名就可以简单地分清层级。

```
require hoge.piyo.foo.bar;
```

上面这段代码，会以 DKM_LOAD_SEARCH_PATH 中设置的目录为起点，以包名的最后一个名字之外的部分（即上述例子中的“hoge/piyo/foo”）为目录进行搜索。总之和 Java 一样，Diksam 的包层级也要和目录层级一致。

另外，在 C++ 或者 C# 中，一个源文件可以对应多个 namespace，但是一般情况下都不会这么做。我觉得在这种事情上节省没有任何意义。相反，一个包可能希望由多个源文件构成。我想就像前面说到的，这是能够使用 Diksam 编写大程序的一个先决条件，也正因如此，我们才需要用最简单的办法来解决眼前实现和使用上的问题。

另外，像 print() 这样标准的程序库被收录在了 diksam.lang 中。在这次要制作的 Diksam 的版本（Diksam book_ver.0.3）中，使用者必须要手动进行 require。

4. rename

在进行 require 时，如果引入了多个包中的同名函数时会发生命名冲突。

在 Java 中，可以通过指定**全限定类名**（FQCN，Fully Qualified Class Name）避免这个冲突（如 java.util.List 和 java.awt.List）。但是，这样编写代码时会很麻烦，而且编写出来的代码也会显得杂乱无章。

因此，在 Diksam 中没有指定 FQCN 的方法。解决冲突的方法是利用 rename 进行如下操作。

```
rename com.kmaebashi.util.print myprint;
```

上面这段代码将 com.kmaebashi.util 包的 print 函数改名为 myprint。

rename 必须写在源代码的开头和 require 之后。还有，rename 的有效范围仅在当前源文件中，即使在被 require 的文件中使用了 rename，也不会影响到进行 require 的文件。这是因为，被改名后的名字只在当前源文件内可见，这样做的目的是为了可以让每个源文件都能识别出函数的真正身份。

5. 开始执行

在现在的 Diksam 中，如果执行下面这段代码，程序将从 hoge.dkm 的顶层结构开始执行。

```
% diksam hoge.dkm
```

即使引入了 require，这个设计仍然没有改变。总之，程序总是会从指定源代码的顶层结构开始执行。一旦程序开始执行，就可以调用被 require 的源代码中的函数了。即使被 require 的文件有顶层结构，也是不会执行的。

可以把 Diksam 的顶层结构看作是 Java 中的 main() 方法。main() 方法在

程序库中大多是充当测试驱动的角色吧[①]。

6. 关于全局变量

在 Diksam 中，全局变量是在函数外（顶层结构中）声明的变量，但是其他源文件是引用不到这个全局变量的。也就是说，如果只有在多个源文件中能够被任意引用的变量才可以称为全局变量的话，那么 Diksam 中就不存在全局变量。

但是，多个源文件之间可以进行函数调用，因此使用 get_xxx()、set_xxx() 也是可以访问全局变量的。一般来说，应该尽可能不使用全局变量，因此我认为这种方式再适合不过了。

补充知识　#include、文件名、行号

虽然和本节的主题无关，但这里还是要提一下，在 8.1.1 节的开头写道：

> **分割源代码的方法，最简单的就是和 C 语言里面的 #include 一样，嵌入来自于其他源文件的代码。这个方法既简单又直接，非常实用。**

这个方法非常实用，那么这个方法就是个好方法吗？你可能会认为，像 C 语言的预处理那样，如果事先进行了处理，编译器就无需在执行时再进行校正了。实际却不是这样的。值得注意的一点就是，必须要通过某种方法知道被 require 的文件的文件名和行号。

使用 #include 将其他文件嵌入进来的话，行号自然会发生变化。在出现报错信息的时候，将变化后的行号输出给使用者的行为是很不友好的（JSP，即 Java Server Pages，它就是这样，会将自动生成的 Java 代码的行号直接输出）。

例如，C 语言的预处理会通过下面的形式，将行号和文件名传递给预处理后的文件 *。

* gcc 好像采用了另外的方式输出。

```
#line 2 "hello.c"
```

8.1.2 DVM_ExecutableList

一个 Diksam 编译器和一个源文件会生成出一个 DVM_Executable。在之后可能会对一个源文件进行分割，因此编译后也可能生成多个 DVM_Executable。

为了管理这些 DVM_Executable，我们引入了 DVM_ExecutableList 结构体（DVM_code.h）。

```
typedef struct DVM_ExecutableItem_tag {
```

① 这也就说明了为什么被 require 的程序不执行其顶层结构的原因。——译者注

```
    DVM_Executable *executable;
    struct DVM_ExecutableItem_tag *next;
} DVM_ExecutableItem;

struct DVM_ExecutableList_tag {
    DVM_Executable      *top_level;
    DVM_ExecutableItem  *list;
};
```

这是一个通过 DVM_ExecutableItem 保存 DVM_Executable 的链表的类。成员 top_level 在通过 list 保存了 DVM_Executable 的同时，也保存了顶层结构（编译器启动时设定的）。

8.1.3 ExecutableEntry

如前面所述，Diksam 中没有跨文件的全局变量。函数外声明的变量被保存在独立的命名空间中，没有进行链接。

在以前的数据结构中，全局变量运行时的内存空间保存在 DVM_VirtualMachine 中，如下所示。

```
typedef struct {
    int         variable_count;
    DVM_Value   *variable;
} Static;

struct DVM_VirtualMachine_tag {
   （中间省略）
    Static      static_v;
   （中间省略）
};
```

但是，正因为没有进行链接，所以对于 DVM 来说（全局变量）没有必要保存为一个数组。一个 DVM_Executable 中保存一个数组就可以了。

在和编译器共用的 DVM_Executable 中，不能只保存运行时使用的数据，因此引入了 ExecutableEntry 结构体，如下所示（dvm_pri.h）。

```
struct ExecutableEntry_tag {
    DVM_Executable   *executable;
    Static      static_v;  ←函数外声明的变量所使用的内存空间
    struct ExecutableEntry_tag *next;
};
```

运行时，只为每个 DVM_Executable 分配一个 ExecutableEntry 数据结构。如上所述，其中保存着之前在 DVM_VirtualMachine 中保存的全局变量（函数外声明的变量）的内存空间。

8.1.4 分开编译源代码

接下来要解决的问题是，在一个源文件使用 require 请求了其他源文件的情况下，如何进行编译比较好？

比较直接的想法，我想是在解析器发现 require 的时候递归调用编译器。但是，因为在 yacc/lex 的内部使用了很多全局变量，所以不能使用递归。也就是说，在解析一个文件的中途不能再去解析其他文件 *。

* 使用 bison 等语法解析器时可以使用递归。

Diksam 的编译顺序是，在一个源文件完成编译后，再按顺序编译被 require 的源文件。

DKC_Compiler 结构体作为 Diksam 编译的核心，其内部保存着顶层结构的语句列表和函数（FunctionDefinition）的列表等。在编译的最后阶段，会根据这个结构体生成 DVM_Executable。从结构上来说，DKC_Compiler 和 DVM_Executable 是 1:1 的关系，因此在源文件中编译被 require 的源文件时，会为其创建一个新的 DKC_Compiler 结构体。

编译 Diksam 的代码时，应用程序会调用 DKC_Compile() 函数。这个函数会调用 do_compile() 方法，如下所示。

```
yyin = fp; /* 将源代码 fg（作为起点）赋值到 yyin 中 */
/* 生成空的 DVM_ExecutableList */
list = MEM_malloc(sizeof(DVM_ExecutableList));
list->list=NULL;

/* 调用 do_compile()。第三个参数为源文件的路径，但在这个层级不适用。 */
exe = do_compile(compile, list, NULL, DVM_FALSE);
```

do_compile() 的内容如代码清单 8-1 所示（节选）。

代码清单 8-1 do_compile()

```
static DVM_Executable *
do_compile(DKC_Compiler *compiler, DVM_ExecutableList *list,
           char *path, DVM_Boolean is_required)
{
    （省略局部变量的声明部分）
```

```
 6:     /* 在 C 的栈中回避当前编译器 */
 7:     compiler_backup = dkc_get_current_compiler();
 8:     dkc_set_current_compiler(compiler);
 9:
10:     /* 执行解析 */
11:     if (yyparse()) {
12:         fprintf(stderr, "Error!Error!Error!\n");
13:         exit(1);
14:     }
15:
16:     /* 遍历所有被 require 的源文件 */
17:     for (req_pos = compiler->require_list; req_pos;
18:          req_pos = req_pos->next) {
19:         /* 检查正在编译的源文件是否有相应的编译器 */
20:         req_comp = search_compiler(st_compiler_list, req_pos->package_name);
21:         if (req_comp) {
22:             compiler->required_list
23:                 = add_compiler_to_list(compiler->required_list, req_comp);
24:             countinue;
25:         }
26:         /* 如果没有，创建一个新的编译器并进行编译 */
27:         req_comp = DKC_create_compiler();
28:
29:        （中间省略。这里将搜索到的源文件路径设置到 found_path 中。）
30:         req_exe = do_compile(req_comp, list, found_path, DVM_TRUE);
31:     }
32:
33:     dkc_fix_tree(compiler);
34:     exe = dkc_generate(compiler);
35:     if (path) {
36:         exe->path = MEM_strdup(path);
37:     } else {
38:         exe->path = NULL;
39:     }
40:
41:     exe->is_required = is_required;
42:     if (!add_exe_to_list(exe, list)) {
43:         dvm_dispose_executable(exe);
44:     }
45:
46:     /* 从备份中恢复当前编译器 */
47:     dkc_set_current_compiler(compiler_backup);
48:
49:     return exe;
50:}
```

在函数中，yyin 的值被设置为源文件的文件指针，并在第 11~14 行解析这个

文件。之后，按照顺序循环地编译从第 17 行开始的代码。

前面已经提到过，当前版本的 Diksam 会为每个源文件生成一个编译器（DKC_Compiler）。然后，执行过一次编译的编译器被保存在名为 st_compiler_list 的 static 的链表中。在第 20 行被调用的函数 search_compiler() 是为了在第 1 个参数 st_compiler_list 中搜索是否有编译过当前包的编译器。DKC_Compiler 保存着包名，如果包已经被编译过，将对应的编译器添加到正在编译的编译器的 require_list 中（第 22~23 行）。这个结构很像树结构，但是这个结构是为了共享同一个被多次 require 的源文件，因此实际上它并不是树结构，而是 DAG（有向无循环图，Directed Acyclic Graph）结构。

如果有没有编译过源文件的话，从第 27 行开始的代码会创建一个新的编译器并进行编译。

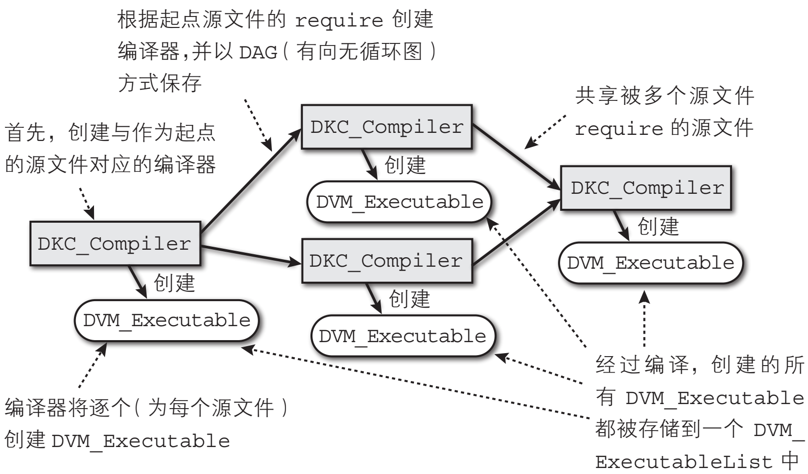
并且，第一次启动时，st_compiler_list 的生命周期是从第一次编译结束到开始运行之前。因此，在动态加载的时候同样的源代码会被再次编译。详细请参考 8.1.5 节的补充知识。

第 29 行中间省略了通过 DKM_REQUIRE_SEARCH_PATH 搜索被 require 的源文件并返回文件指针等一连串的处理。之后将返回的文件指针作为参数，递归调用 do_compile() 函数（第 30 行）。

所有的子文件都编译过后，将使用 dkc_generate() 创建 DVM_Executable，然后将其注册到 DVM_ExecutableList 中（第 42 行）。

最终状态的 DKC_Compiler 的结构如图 8-1 所示。

图 8-1
编译器编译后的结构



8.1.5 加载和再链接

由于编译是以源文件为单位进行的，因此完成编译之后，还需要进行链接。**链接**是指把不同源文件中出现的同名函数进行对应的操作。本节是继续 6.4.1 节的内容作介绍。

在 C 等语言中，全局变量也必须进行链接，但是在 Diksam 中并没有可以跨源文件的全局变量，因此有必要和其他源文件进行链接的就只有函数了（虽然后面会出现类的概念）。

现在，“函数”的数据保存在以下三个地方。

1. DKC_Compiler中的FunctionDefinition列表

这个对应表中保存了在当前源文件中定义的原型声明函数。如果只是签名声明的话，指向实现程序块的成员 block 应为 NULL。

这其中并不保存被 require 的 .dkh 文件中声明的函数。这些函数将保存在自己所在 .dkh 文件对应的编译器中，因此，在搜索函数的工具函数 dkc_search_function() 中，为了搜索参数指定的函数，需要递归地遍历所有子编译器（util.c）。

2. DVM_Executable中的DVM_Function数组

DVM_Executable 的 DVM_Function 数组中，保存着当前源文件中出现的所有函数。

假设在源文件 a.dkh 中 require 了 b.dkh，并且调用了 b.dkh 中声明的函数 b_func()。此时，b_func() 并没有保存在 a.dkh 的 FunctionDefinition 中，而是保存在 DVM_Executable 中。

就像 6.4.1 节中介绍过的那样，在完成编译时，指令 push_function 指定的函数的索引值就是这个数组的下标。因此，一般情况下所有需要被调用的函数都会保存在这个对应表中。

至于索引值，加载时会在字节码中直接替换为 DVM_VirtualMachine 中 Function 数组的下标（请参考 6.4.1 节）。

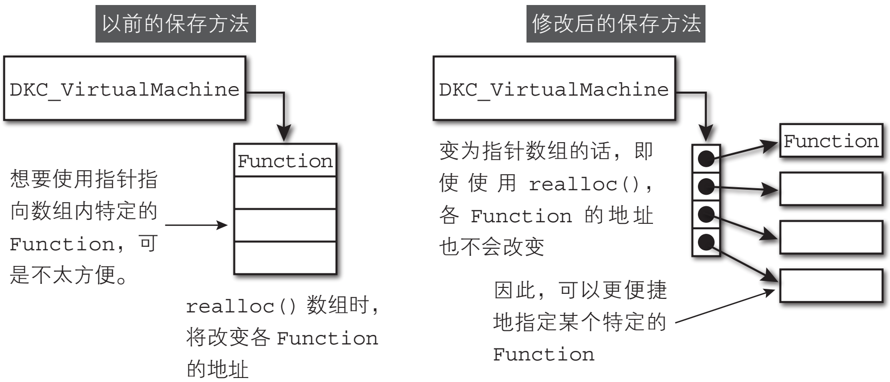
3. DVM_VirtualMachine 中的 Function 数组

DVM_VirtualMachine 中的 Function 数组保存着链接后的函数，每个 DVM 中只有一个该数组。

因为 Diksam 是动态加载的，所以这个对应表中保存的函数还是没有实现的状态。此时，成员 is_implemented 都是 false。

并且，这个数组在以前的版本中以可变长数组的形式保存在了 DVM_VirtualMachine 结构体中，但是，现在变成了“指针的可变长数组”。后面将会说到，基于动态加载，这个数组会使用 realloc() 进行扩展。之前的结构如果使用了 realloc() 会使 Function 结构体的地址发生变化。因此，我们需要一个即使扩展了数组的元素地址也不会改变的结构（如图 8-2）。

图 8-2
修改为即使使用了 realloc() 元素，地址也不会改变的结构



具体的实例请参考以下三段源代码（hoge.dkm，piyo.dkh，piyo.dkm）。

hoge.dkm:

```
require piyo;

int hoge_func() {
    piyo_func();
}

hoge_func();
```

piyo.dkh:

```
int piyo_func();
```

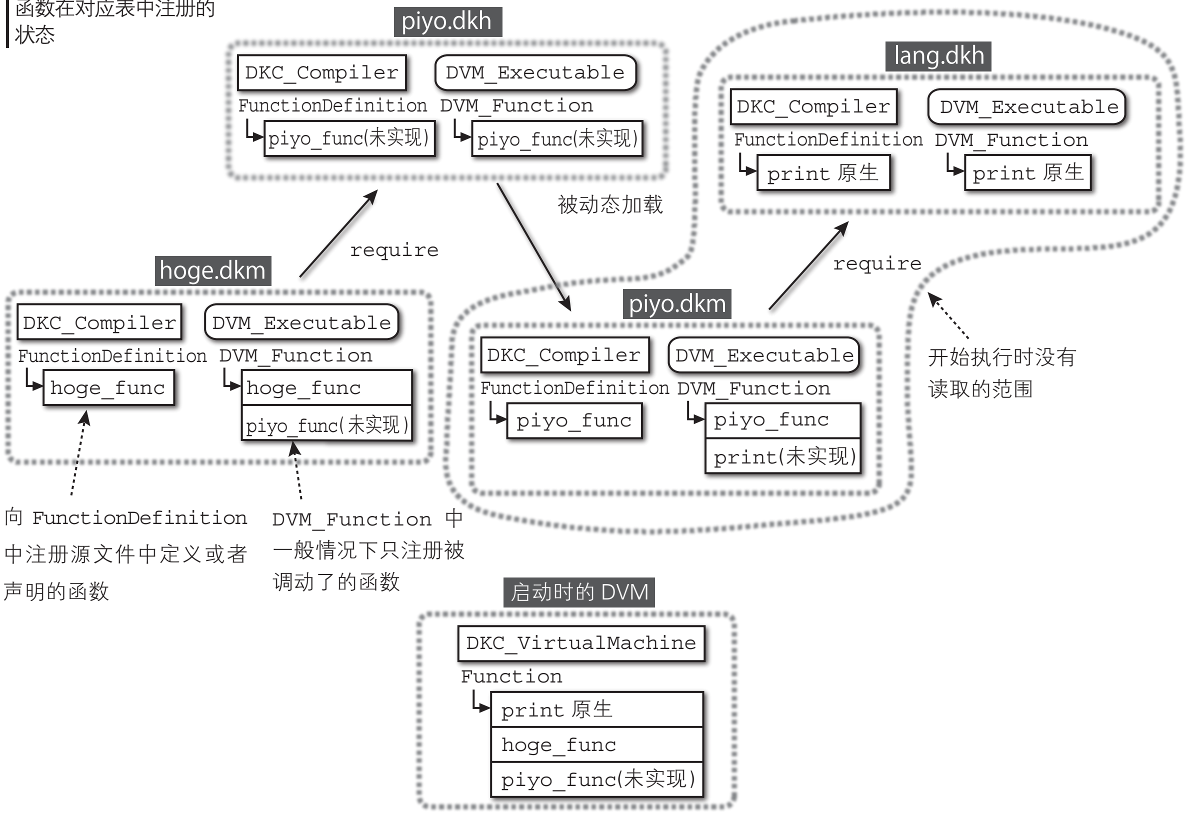
piyo.dkm:

```
require diksam.lang;

int piyo_func() {
    print("piyo piyo\n");
}
```

各个源文件在对应表中的注册状态如图 8-3 所示。

图 8-3
函数在对应表中注册的状态



由于在 hoge.dkh 中仅定义或者签名声明了函数 hoge_func()，因此 FunctionDefinition 也只注册了一个函数。但是，DVM_Executable 的 DVM_Function 中只注册了被调用的 piyo_func() 函数。push_function 指令创建不了的函数不会注册到这里。

hoge.dkm 中被 require 的 piyo.dkh 也会被同时编译。在这里被声明的 piyo_func() 会同时注册到 piyo.dkh 的 FunctionDefinition 和 DVM_Executable 两个地方。

在执行开始时只编译了这两个文件。在这个状态中，会根据 DVM_VirtualMachine 的 Function 创建对应表，并将经常以原生函数形式出现的 print()、hoge_func() 和 piyo_func() 注册进来。但是，虽然 piyo.dkm 描述了实现但还没有被加载，因此还没有实现 piyo_func()。

一旦 piyo_func() 被调用，就会进行动态加载。动态加载功能首先新创建一个 piyo.dkm 的编译器，然后再创建一个被 require 的 lang.dkh 的编译器，最后将创建出来的 DVM_Executable 链接到 DVM_VirtualMachine 中，并开始执行 piyo_func()。

补充知识　动态加载时的编译器

只需一次编译，就可以让同一个文件在任何地方都能被 require，与此相对，只创建一个 DKC_Compiler 就可以了。在代码清单 8-1 中说明了其实现方法，即使用 static 的变量 st_compiler_list 保存所有编译器。

但是，也会发生这样的情况，在 a.dkm 中 require 了 b.dkh，b.dkh 中动态加载的 b.dkm 又调用了 a.dkm 中的函数。在现在的实现中，只会为 a.dkm 创建一个 DVM_Executable，但是却会多次创建 DKC_Compiler。a.dkm 的 DKC_Compiler 在第一次编译后将被销毁，但编译 b.dkm 时又必须使用 a.dkm 的编译器。在 b.dkm 的 FunctionDefinition 中只注册了 a.dkm 中被调用的函数，但请记住它是会搜索子编译器的。

同一个源文件被编译多次确实很浪费，但是现在的 Diksam 还不能把字节码保存在文件中，因此在执行时必须要有源代码。考虑到这样的情况，虽然很浪费但也没有什么坏处，在实用性上也不会有问题。

等到日后字节码可以保存到文件中的时候，只需使用 DVM_Executable 中包含的 DVM_Function 等信息就可以完成源代码的编译了。



8.2 设计 Diksam 中的类

在成功将源文件分割后，接下来就要考虑类的设计了。

8.2.1 超简单的面向对象入门

设计类一定会考虑到数组（array）。但是，考虑到本书的目标读者是掌握 C 语言，并具有一定代码阅读能力的程序员，因此，我觉得这里突然开始类和面向对象的话题好像不太合适。另外，面向对象的相关用语在不同语言中也不尽相同 *，因此，本节将要讲解的是包含用语含义在内的一些简单的面向对象概念。

* 例如 Java 中的“超类”在 C++ 中被称为“基类”。

Diksam 的面向对象与 Java、C++ 和 C# 相同，都是**基于类的面向对象**。类



（class）近似于C中的结构体类型，但是与其关联的动作（函数）可以保存在类中，被称为**方法**（method），数据成员被称为字段（field）。

```
class Point {
    double x;
    double y;
    // 定义 Point 的方法 print()
    void print() {
        println("(x, y)..(" + this.x + ", " + this.y + ")");
    }
}
```

方法以 `p.print()` 的形式调用。如果是C语言的话，第1个参数要传入指向 `Point` 的指针。但是，在面向对象的语言中，每个类都拥有不同的命名空间，其优势在于（与C语言相比）无需特别注意命名。

之前说到了类“近似于C中的结构体”，在C语言中声明结构体的类型时，并不会为其创建内存空间。与此相同，Diksam中也需要使用 `new` 来创建内存空间，相当于C语言中的 `malloc()` 操作。被 new 创建出来的叫作**对象**（object）或者**实例**（instance）。

```
Point p = new Point();
```

Diksam中的类全部属于引用类型，因此上面代码中的 `p` 相当于C语言中的指针。但是，如果要引用字段或者方法，不是使用 `->` 而是使用 `.`（和Java等语言相同）。

在Diksam这样的面向对象语言中，可以使用**继承**（inheritance）的方式为类添加字段或者方法。例如，在制作一个二维图形绘制工具时，要定义一个代表图形的类 `Shape`。在 `Shape` 中保存着“颜色”等所有图形都共有的属性。又如，“开始和结束坐标”是直线（Line）中特有的数据。因此，如果在定义 `Line` 的时候继承 `Shape`，那么 `Line` 将既具有 `Shape` 的“颜色”属性，又具有自己特有的“开始和结束坐标”属性。

与此相似，crowbar和Diksam的源代码中，在 `Expression` 结构体和 `Statement` 结构体中的实现方法是“使用枚举类型区分不同种类（数据类型），将每个种类的数据保存在联合体中”。可见，在C语言中想要实现这个功能，必须在程序员的层面“约定”，但是，对于面向对象的语言来说，它本身就能够显式地提供此功能。

如此一来，在 `Line` 继承 `Shape` 的时候，`Shape` 被称为 `Line` 的超

类，Line 被称为 Shape 的子类。根据不同语言超类也叫作**父类**（parent class）和**基类**（base class），子类也叫作**孩子类**（child class）和**派生类**（derived class）。另外也有“把类沿着超类方向追溯到的所有类称为**祖先**（ancestor），并把子类方向的所有类称为**子孙**（descendant）”的说法。

子类的引用通常可以赋值给类型为父类的变量，也就是说 Line 或者 Circle（圆）可以赋值给 Shape。例如，有一个 Shape 类型的数据，其中可以保存 Line、Circle、Rectangle（矩形）等图形。

然后，为了描述数组中的所有图形，为 Shape 添加 draw() 方法（可以不实现）。

```
// abstract 和 virtual 的话题将在后面叙述
abstract class Shape {
    （中间省略）
    // 声明没有实现的 draw() 方法
    abstract virtual void draw();
}
```

在每个子类中**覆盖**（override）这个方法。

```
// 继承了 Shape 的 Line 类的定义
class Line : Shape {
    （中间省略）
    override void draw() {
        //Line 的描绘处理
    }
}
```

基于上面这些代码，数组中保存的 Shape 将以如下方式依次调用 draw() 方法，如果是 Line 的话调用 Line 的描绘方法，如果是 Circle 的话则调用 Circle 的描绘方法。这种特性被称为**多态**（polymorphism）。

```
Shape[] shape_array;
// 假设已经设置过 shape_array 的值了
for (i = 0; i < shape_array.size(); i++) {
    shape.draw();
}
```

与在 crowbar 和 Diksam 中的做法相似，C 语言中如果也使用枚举类型和联合体实现“模拟继承”的话，就必须进行“根据枚举类型用 switch case 判断分支”的处理了（比如 Diksam 的 fix_expression 就包含一个巨大的 switch case）。编写 switch case 本身倒是不成问题，问题在于当枚举类型的种

类增加的时候，就不得不去修改分散于各处的 swith case。特别是 Shape，当增加图形种类的需求越来越强烈的时候，这个问题就变得尤为突出了。在使用了多态后，即使图形的种类增加了，在上述示例代码中也没有必要修改 Shape 的 draw() 方法的调用位置，也就不必再次进行编译了 *。

* 因为 Diksam 并没有将字节码保存为文件，所以不论如何都要重新编译。

就像上面说到的，子类的引用通常可以赋值给类型为超类的变量（此处发生的自动类型转换被称为**向上转型**，即 up cast），但是这并不意味着超类的引用能够赋值给子类的变量。Line 必然是 Shape，但 Shape 不一定是 Line（说不定是 Circle 或 Rectangle 呢）。然而，在 Diksam、Java、C#、C++ 中都有强制将 Shape 转换（即**向下转换**，down cast）为 Line 的手段（在向下转换时有可能会发生错误）。

在 Diksam、Java 和 C# 中，除了类之外，还有**接口**(interface) 的概念。它类似于只有方法的声明的类。

例如，Line 和 Circle 可能会有在调试时显示坐标等信息的需求。如果只考虑 Shape 的话，为 Shape 添加 print() 方法并由各图形进行覆盖就可以达到目的了，但是如果考虑到下面的情况呢？

> 想要制作一个 void debug_write (Printable obj) 函数以便控制程序“只在调试模式时输出”。
> 传递给这个函数的参数对象也许和 Shape 之间并没有任何继承关系。

这种情况下就可以使用接口了。

```
interface Printable {
    void print();
}
```

有了上面的接口后，各个类都可以对其进行实现，每个类都会有“print 自己的内容”这个功能的通用接口（实际的编写方法请参考 8.2.4 节）。

Diksam 中的类只能进行单继承（single inheritance），即一个类只能对应一个超类。这种方式在 Java 和 C# 中相同。但是，在 C++ 中是允许**多继承**（multiple inheritance）的。

关于接口，Diksam 是允许多继承的。这一点也和 Java、C# 相同（C++ 允许类的多继承，因此一般情况下不会出现接口）。

基于类的面向对象大概就介绍到这里，没有介绍到的部分请大家参考市面上其他的参考书吧。

8.2.2 类的定义和实例创建

Diksam 中类的设计基本上采用了 C++、Java、C# 的方式，虽然更像是由 C++ 派生出的类型的语言，但是也有意地改变了一些地方。

作为类定义的典型例子，首先让我们考虑一下在二维坐标上保存一个点的类 Point（代码清单 8-2）。

代码清单 8-2
Point 类的定义

```
 1: require diksam.lang;
 2:
 3: public class Point {
 4:     private double x;
 5:     private double y;
 6:
 7:     double get_x() {
 8:         return this.x;
 9:     }
10:     void set_x(double x) {
11:         this.x = x;
12:     }
13:     // 由于篇幅限制，省略get_y()，set_y()
14:
15:     void print() {
16:         println("x.." + this.x + ",y.." +this.y);
17:     }
18:     // 默认的构造函数的名称是“initialize”
19:     constructor initialize(double x, double y) {
20:         this.x = x;
21:         this.y = y;
22:     }
23: }
24:
25: // 创建 Point 的实例
26: Point p = new Point(10, 20);
27:
28: // 显示 p 的内容
29: p.print();
```

虽然看上去和 Java 等语言的类差不多，但是 Diksam 的类有以下这些不同点。

1. 引用成员时必须使用 this.

看了第 8 行应该能明白，作为返回值返回成员变量（字段）x 时写作 this.x。在 Java 等语言中虽然可以用这种写法，但如果只写 x 也可以引用到这个字段。可是在 Diksam 中，不显式地写上 this. 的话就不能引用属于类自身的字段和方

法。这里是有意为之。

类是多个方法的集合体，有时候也可能会制作一个相当巨大的类。虽然在 Java 等语言中的做法是为了能够简单地引用到字段，但是随着类的增大，字段会慢慢呈现出全局变量的样子。

例如。`java.awt.Component` 的源代码大约有 8500 行（JDK5.0），在其中能够自由地引用像 `x`、`y` 这么短名字的属性，这对我来说简直就是自杀行为。

因此在 Diksam 中，即使是在类内部引用类的成员，也必须使用 `this.`。

这里采用了和 Python 一样的语法，可能有些人讨厌这样，但是我挺喜欢的。

2. 成员的访问修饰符只有`public`和`private`，没有 `protected`

在 Diksam 中 `public` 是从包外部可以访问的意思。第 3 行的 `class` 前面附加的 `public` 就是这个意思。第 4~5 行为成员设定的 `private` 是不能从本类以外访问的意思。如果同时加上 `public` 和 `private` 的话，就只有当前包内可以引用。

而且，通过代码清单 8-2 就应该能理解，在 Diksam 中类的访问修饰符 `protected` 是不存在的。这样做的原因将在后面详细介绍，但是基本的思路是，不要让因为别人有可能会制作子类，成为放松访问限制的借口。

3. 构造方法要使用`constructor`修饰符

构造方法（constructor）是在创建类的实例时调用的方法，通常会在里面编写类的初始化处理。

Java、C++、C# 等语言的构造方法必须和类名相同，可以使用**方法重载**（method overload）实现多个构造方法。方法重载就是多个方法名称相同，但参数的数量和类型不同，内部处理也不同（但编程语言会认为它们是一个函数）*。

但是，方法重载是混乱的根源 *，毕竟不是什么都能用重载来解决的。例如，在代码清单 8-2 的 `Point` 类中，将直角坐标系的 `x,y` 传递给构造方法。但是如果是使用极坐标的应用程序，恐怕就要给构造函数传入 θ（偏角）和 ρ（极径）了。但不论是直角坐标系的 `x,y`，还是极坐标的 θ、ρ，都是两个 `double` 的组合，因此在方法重载上无法区分（这个例子中指出了后面将要介绍的 OOSC[8] 的问题）。总之，"构造方法要和类的名字相同"这种设计本身，我认为是不合理的。

* 重载很容易和重写混淆，它们是两个完全不同的概念。

* 如果遇到了 `int` 升级为 `double`（译注：构造方法在同一位置上的参数既有 int 型参数的也有 `double` 型参数的）或者继承[①]的情况，就不能简单地决定到底使用哪个方法了。

① 构造方法在同一位置上的参数既有子类型的参数也有父类型参数。——译者注

因此在 Diksam 中为构造方法加上了 `constructor` 关键字，在 `new` 的时候可以指定任意的构造方法。

代码清单 8-2 的第 26 行省略了方法名。这种情况下，会直接调用默认构造方法 `initialize()`（第 19 行）。如果第 26 行写成下面这样：

```
Point p = new Point.myinit(x, y);
```

就会直接调用被指定的 `myinit()` 方法代替原来的 `initialize()` 方法。

在定义类的时候，如果没有定义任何的构造方法的话，编译器会自动添加一个默认构造方法（default constructor），如下所示：

```
public virtual override constructor initialize() {
    super.initialize(); ←只有在有超类的时候才有这行语句
}
```

4. 没有static的字段和方法

在 Java、C++ 和 C# 中，使用 `static` 关键字可以定义一个与实例无关的字段或者方法。与实例无关的意思就是，除了访问域的问题外，其他方面与全局变量或者函数完全相同。在 Diksam 中一般都会写成全局变量或者函数，因此抛弃了 `static` 的字段或者方法。

8.2.3 继承

说起类当然会提到继承。代码清单 8-3 就是 Diksam 中继承的示例代码。`Point2` 继承了 `Point1` 并重写了 `print()` 方法。

代码清单 8-3
Diksam 中的继承

```
require diksam.lang;

// 不使用 abstract 关键字的类不能被继承
abstract class Point1 {
    double x;
    double y;

    // 不使用 virtual 关键字的方法不能被重写
    virtual void print() {
        println("x.." + this.x + ", y.." + this.y);
    }
    // 构造方法不使用 virtual 也不能被重写
    virtual constructor initialize(double x, double y) {
```

```
14:         this.x = x;
15:         this.y = y;
16:     }
17: }
18:
19: // 继承时没有使用 Java 的 extends 关键字，而是使用了 C++/C# 的 ":"
20: class Point2 : Point1 {
21:     // 进行重写的时候要使用 override 关键字
22:     override void print() {
23:         println("override: x.." + this.x + ", y.." + this.y);
24:     }
25:     // 构造方法也可以被继承和重写
26:     override constructor initialize(double x, double y) {
27:         this.x = x + 10;
28:         this.y = y + 10;
29:     }
30: }
31:
32: // 给 Point1 的变量 p 赋值为 Point2 的实例
33: Point1 p = new Point2(5, 10);
34:
35: // 由于方法被重写了，所以调用的是
36: // Point2 的 print() 方法
37: p.print();
```

代码清单 8-3 的执行结果如下：

```
overrided: x..15.000000, y..20.000000
```

通过这样一个结果，能看出如下特征：

1. 方法默认为 non virtual

在第 8 行为想要被重写的 `print()` 方法添加了 virtual 关键字。在 Diksam 中类会被默认定义为不可被继承（在 Java 中为 final）的，只有显式地添加关键字 virtual，方法才可以被继承。这里的设计和 C++、C# 一致。

关于这点我想还存在很多异议，详细内容我会在后面的章节中说明。

2. 重写时必须使用`override`关键字

第 21 行，Point2 重写 Point1 的 `print()` 方法时在前面添加了 override 关键字（与 C# 相似）。如果不加 override，又定义了和超类同名的方法的话，就会发生错误。

这点的根据是“重写必须显式进行”的原则，还得到了一个附带的好处，就是如果误定义了函数名，会发生编译错误。

3. 使用：继承

在第 18 行中定义了继承 `Point1` 的 `Point2` 类。之所以没有使用 Java 的关键字 `extends`，是因为不想让继承的关键字太长，因此使用了 C++/C# 的“:”。

4. 构造方法也可以被继承和重写

在 Java、C++、C# 中，构造方法都是与类名相同的，因此构造方法不能被继承（但是可以通过 `super()` 调用）。另外，从语法角度讲也不可能重写构造方法。

但是，在 Diksam 中可以任意决定构造方法的名字，我认为能够重写构造方法也不是坏事 *。因此，在 Diksam 中构造方法也可以被继承和重写。基于这个设计，就可以把在超类中定义的构造方法看作是默认实现。

* 实际在 Eiffel 中构造方法也是可以重写的。

5. 只有 abstract 的类可以被继承

第 4 行，为类 `Point1` 添加了 `abstract` 关键字。添加了 `abstract` 的类（抽象类）只是为了被继承而存在的类，抽象类本身不能被实例化。

因此，在 Diksam 中抽象类以外的类（**具象类**，concrete class）不能被继承。

对于这个限制，可能大多数人会持否定的态度。但是，这个设计并不是为了让实现起来更简单而妥协的结果，是有意为之。至于为什么要这么做，我会在后面的章节中说明。

8.2.4 关于接口

与 Java、C# 一样，Diksam 的类也只能单继承。能够多继承的只有接口（代码清单 8-4）。

代码清单 8-4
接口的定义

```
require diksam.lang;

interface Printable {
    void print();
}

class Point : Printable {
    double x;
    double y;

    override void print() {
```

```
        println("x.." + this.x + ", y.." + this.y);
    }
    constructor initialize(double x, double y) {
        this.x = x;
        this.y = y;
    }
}

Printable printable = new Point(10, 20);

printable.print();
```

在这个例子中，定义了一个具有 print() 方法的接口 Printable，并且在 Point 类中对其进行了实现。

在多继承的时候，使用逗号分隔多个接口名。此处的设计和 C# 一样（其实，除了没有 implements 关键字之，其他的和 Java 都差不多），没有什么特别的创新。

但是，在 Diksam 中，接口是不能继承接口的。因为接口间的继承通过在实现类中继承多个接口就能达到一样的效果，所以这在现阶段还不是必须要做的事情。

8.2.5 编译与接口

在 Diksam 中，如果 require 了只有某个函数签名声明的 .dkh 文件，那么也可以编译通过。接下来，在函数执行的时候会加载定义了其实现的 .dkm 文件，并通过动态编译功能进行编译。

对于类的方法来说，并没有特别地提供这种机制，但是使用接口也可以实现设计和实现的分离。

首先，在 .dkh 文件中实现定义接口和返回接口对象的函数（代码清单 8-5）。

代码清单 8-5
classsub.dkh

```
// 在 .dkh 文件中定义接口
public interface ClassSub {
    public void print();
}

// 定义实现接口的返回接口对象的函数
ClassSub create_class_sub();
```

然后，在对应的 .dkm 文件中，定义实现接口的类以及返回类实例（通过 `new`）的函数（实现在 .dkh 文件中的所有签名声明）。

这样一来，在调用 `create_class_sub()` 函数的时候就会发生动态加载。在此之前，`classsub.dkm` 都不会被加载。

8.2.6 Diksam 怎么会设计成这样？

本节将介绍 Diksam 的面向对象为什么会设计成现在这样。如果只想了解实现方法的人可以跳过本节。

C++ 为我们提供了以下这些不错的参考。

> C++ 的方法默认为 non virtual。这是一个略微提升效率却牺牲了类扩展性的万恶设计。在 C++ 中创建一个类时要尽量在方法前面加上 *protected virtual*。

面向对象的圣经之作 *Object-Oritented Software Construction*（简称 OOSC[8]）中对 C++ 作了如下评价：

> 在进行声明的时候是否必须使用 `virtual` 的意思就是，必须要有明确的约束策略（静态约束还是动态约束）。而这个策略违反了开放 / 闭锁原则。（中间省略）
> C2（不明确标示 "`virtual`" 的时候，默认使用静态约束）更加恶劣，很难看出这种做法在语言设计上的正当性。正如上面说到的，让静态约束和动态约束拥有不同的含义，这个选择本身就是错误的，在此基础上还要进行默认选择，更是错上加错了。

真是个灾难。

在 C++ 之后问世的 Java 改成了默认 `virtual`（在 non virtual 的时候要使用 `final`）。

然而，在 Java 之后出现的 C# 中又改回了默认 non virtual。关于这点，我（通过网络等途径）也听过不少严厉的批评，但 C# 的作者 Anders Hejlsberg 作出了如下回应：

> There are two schools of thought about virtual methods. The academic school of thought says, "Everything should be virtual, because I might want to override it someday." The pragmatic school of thought, which comes from building real applications that run in the real world, says, "We've got to be real careful about what we make virtual."

> 翻译：
> 在 virtual 方法的问题上存在两派。学术派主张“任何事物都应该是 virtual 的。因为有一天我可能需要重写它”。与此相对，实用派主张构建在真实世界中运行真实的应用，他们认为“在进行 virtual 的时候，本来就应该引起注意”。

说到这里我想起来，《OOSC》的作者 Bertrand Meyer 是大学教授。

方法的重写，替换了一部分在超类中实现的行为。如果不是从一开始就决定了要替换的目标方法，而是随意地重写方法的话，那么在之后超类进行升级等改变时，就无法保证子类的正常运行。总而言之，应该慎重对待这种随便给别人的类打补丁的行为。

基于以上这些，更进一步地让我有了“一般情况下，类应该默认为不可继承”的想法。

下面我要介绍一下在这个问题上“实用派”的一些想法。

《Effective Java 中文版（第 2 版）》[9] 第 15 章中“要么专门为继承而设计并给出说明文档，要么禁止继承”说道：

> 这不仅仅是理论性的问题，如果不是专门为继承而设计并给出相应文档，又非 `final` 的具象类，一旦修改了内部，就要承受与其所有子类关联的地方都有可能发生 bug 的后果。
> 这个问题的最佳解决方案是，对于那些并非为了安全地进行子类化而设计和编写文档的类，禁止其子类化。

《设计模式：可复用面向对象软件的基础》[10] 日文第 1 版 p.31

> 继承使子类获得了父类里实现的详细内容，因此说“继承破坏了封装性的概念”[Syn86]。子类的实现和父类的实现有着紧密的联系，因此改变父类的实现会对子类产生非常强烈的影响。
> （中间省略）
> 对于这点有一个挽救的方法，就是只继承抽象类。

对于“只继承抽象类”这个方法，其实在 Diksam 中已经实现了。

举个例子，在 UNIX 的经典 GUI 工具包 Motif 的类层级中，`PushButton`（常被称为 GUI 的按钮）是 `Label` 的子类。Label 具有显示一个字符串的功能，在定义了继承 Label 的 `PushButton` 时，可以复用 `Label` 中的一些实现。

但是，后来出现的 Java 的 AWT 中，`Label` 和 `Button` 之间就不存在继承关系了。并且，在 Swing 中作为 `JButton`、`JMenuItem`、`JToggleButton` 的超类都引入了 `AbstractButton` 抽象类。

这样虽然会减少复用，却让随意地继承 Label（像 Motif）的方式成为了过去。如果有通用的部分，就明确地定义抽象类，这种做法可以说是非常现代化的方式。

从我的自身经验来讲，从零开始设计的部分，我从来没有想过要它继承具象类。实际上在使用现存的 C++ 类库 * 时，会有“这个方法如果是 virtual 的就要重写”的想法。这样一来，（对头文件来说）直接参考源代码去研究重写的时候，会在不经意间陷入程序库的实现中。这与“把字段写成 public”的想法一样，属于不健康的想法。

* 具体来说就是 MFC（Microsoft Foundation Class）。

顺着这个思路，那么“仅子类可以访问”这个访问修饰符 protected 也可以不需要。

至少在 C++、Java 和 C# 的思路中，访问修饰符是为了对自己（或自己的团队）之外的开发人员隐藏实现而使用的（应该可以作为佐证的是，如果是同一个类，那么即使是不同的实例，也可以相互引用 private 成员）。如果是这样的话，也就没有理由减弱子类的访问限制了。但如果是自己创建的子类的话，可以在包范围内进行访问（与 Java 一样，Diksam 也将包范围作为默认项）。如果想要别人也可以创建子类的话，那就必须要公开了 *。

* 在 Template Method 模式中经常会用到，为了显式地表现出“这里有需要在子类中修改的地方”从而使用 protected，但如果只是用在这种用途上，我觉得还不如使用 public。在使用设计模式的时候，一般情况下必须要有设计文档。

本节虽然介绍了不少内容，但仅代表我个人观点。而且我一直认为自由地决定设计方案是语言作者的特权，因此大家在将来制作属于自己的编程语言时，也可以采用你们自己认为最好的方式。

8.2.7 数组和字符串的方法

在 7.3.4 节中介绍过，在 Diksam book_ver.0.2 中没有取得数组大小和字符串长度的方法（method）。在引入了类的概念后，现在我们来实现这个方法。

要实现的方法如表 8-1 所示。

表中的 T 表示数组元素的类型，也就是说，可以使用 insert() 插入使用代码 double[] a; 声明的数组，但只能是 double 型（这是当然的啦）。

表 8-1 数组和字符串的方法

目标	返回值类型	方法名称和参数	功能
数组	int	size()	取得数组的元素数量
数组	void	resize(int new_size)	改变数组的大小
数组	void	insert(int pos, T item)	向数组中间插入元素

（续）

目标	返回值类型	方法名称和参数	功能
数组	void	remove(int pos)	删除数组的元素
数组	void	add(T item)	向数组末尾添加元素
字符串	int	length()	取得字符串长度
字符串	string	substr(int pos, int len)	截取从 pos 开始长度为 len 的字符串

但由于在 Diksam 中数组和字符串并不是类，因此不能创建出继承它们的类。

8.2.8 检查类的类型

Diksam 和 Java 一样使用 instanceof 运算符。

instanceof 运算符用于判断某个实例是否属于某个类。

例如，继承了 Shape 的 Line 和 Circle。

```
Shape shape = new Line();
```

上面的语句给 shape 赋值了 Line 的实例，此时 shape instanceof Line 返回真。当然，shape instanceof Circle 返回假。

但是，虽然 shape instanceof Line 返回真，但是返回真的也不仅限于 Line 的实例。假设 Line 存在子类 ArrowLine，并且 shape 实际也指向了 ArrowLine 的话，shape instanceof Line 还是会返回真。这就是 instanceof 运算符“判断某个实例是否属于某个类”的含义。

instanceof 也可以用于判断实例是否实现了某个接口。obj instanceof Printable 如果为真，说明 obj 实现了 Printable。

8.2.9 向下转型

Diksam 可以将类向下转型。

Java、C# 等语言也可以向下转型，像下面这段代码。

```
Shape shape = new Line();
  ⋮
Line line = (Line)shape;
```

只是，前置这种转换运算符的书写方式，在成员、数组、方法调用堆叠了很

多层的时候，如果要进行转换，视线必须要回到左边。另外，在转型后如果要再次引用其转型后的结果的话，外面必须要加上括号。

```
// 取得 [piyo.foo[i].bar.fuga[j].getObj()] 的对象，
// 向下转型为 Bazz，
// 并取出其成员 hoge。
Hoge hoge = ((Bazz)piyo.foo[i].bar.fuga[j].getObj()).hoge;
```

C# 中存在后置转型运算符 as，但是由于优先级低，因此还是要使用括号。

由于这点不是很方便，因此在 Diksam 中使用了后置的转型运算符。书写方式为 :: 类型名 :>。上面的例子在 Diksam 中写成了如下代码。

```
Hoge hoge = piyo.foo[i].bar.fuga[j].get_obj()::Hoge:>.hoge;
```

另外，也可以实现从接口向下转型到接口。下面的例子中，Hoge 在实现了 Serializable 的情况下可以成功转型。

```
Printable p = new Hoge();
Serializable s = p::Serializable:>;
```

这种做法在类的继承关系上并不存在“向下”转换。而且这个例子叫作“向下转换”是否贴切也是个问题 *。

* 因此，在例如 Java 的编程语言设计中，不叫作“向下转换”而叫作“缩小转换”(narrowing cast)。但是鉴于大家比较习惯说“向下转换”，因此 Diksam 还是采用了这个说法。

8.3 关于类的实现——继承和多态

实现类最应该关心的不就是继承和多态的实现方法吗？——不管别人是不是这么想，至少我是。

因此，先把其他细节的话题放在一边，本章我们就来介绍一下 Diksam 中继承和多态的实现。

8.3.1 字段的内存布局

首先要考虑的是，在继承了其他类的情况下，字段在内存中的布局。

典型的例子就是，考虑创建类型 Line 和 Circle，并继承表示二维的“图形”的类 Shape。Shape 中保存着“颜色”等图形的通用属性。在 Line 和 Circle 中，也保存着表示各自图形的必备信息。图 8-4 中记载了这些类（图中对方法也进行

了描述，具体会在介绍多态时进行说明)。

图 8-4
“图形”的继承结构

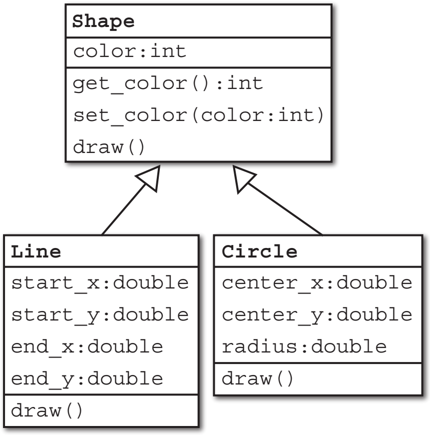
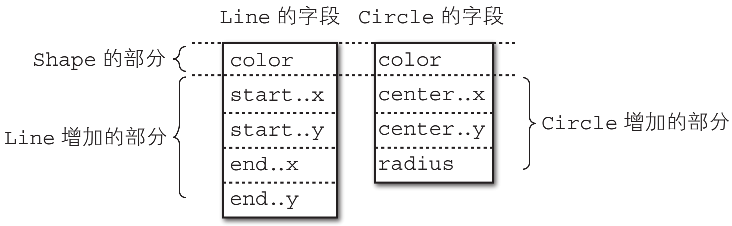


图 8-4 中，Shape 中保存了用 int 型的“颜色编码”，Line 中保存了直线的起点和终点。我想聪明的读者一眼就能看明白，start_x、start_y 是起点坐标，end_x、end_y 是终点坐标，Circle 的 center_x、cengter_y 是原点坐标，radius 是半径。

此时，字段数据的保存方式如图 8-5 所示，超类字段的后面紧跟着子类中增加的字段。

图 8-5
字段的存储方式



使用这种存储方式的好处在于，在引用 Shape 类型变量 shape 的字段 shape.color 时，具体的对象实际上无论是 Line 也好，Circle 也好，它们在引用 color 时的偏移量都是相同的。

如果出现了多继承的话情况就变得复杂了，但是由于 Diksam 的类只能单继承，因此字段的储存也可以使用这种方式实现。

8.3.2 多态——以单继承为前提

继承，不仅要能引用字段，还必须要实现多态。回到图 8-4，Shape 中定义了 get_color()、set_color() 和 draw() 三个方法。其中 draw() 是 abstract 方法，并且会在 Line 和 Circle 中重写。因此，只需编写代码就可以了。

```
shape.draw();
```

此时 shape 不论是 Line 还是 Circle，都会执行 draw() 方法。

限制为单继承的好处就在于容易实现。只需要让类的实例中保存着指向方法实现的指针 * 数组即可。这里说的数组恐怕是原来的 C++ 用语，一般被称为 vtable（virtual method table 的简称）。

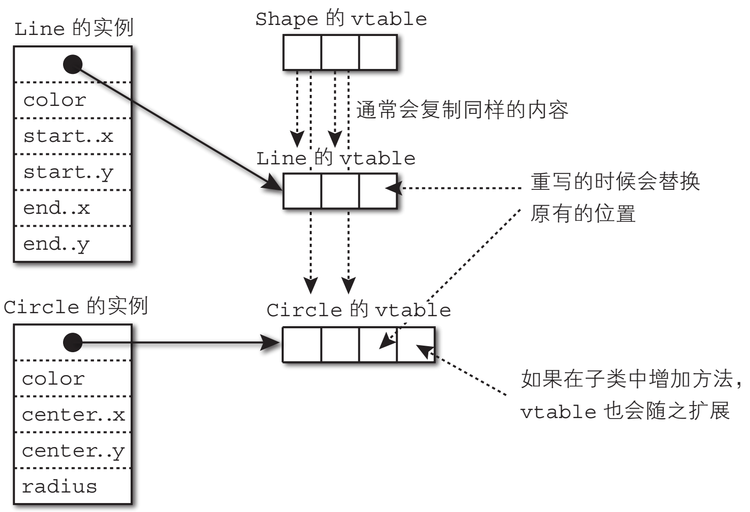
* 这里也不一定非要是 C 语言中所说的指针（内存地址）。在 Diksam 中就是 DVM_Virtual-Machine 中保存着的 Function 表的下标。

vtable 与类相对存在，同一类的实例指向同一个 vtable，并在 new 的时候将 vtable 传递给实例。由于 Shape 包含了三个方法，因此我们通过下面的列表来调用 Shape 的方法。

- get_color() 在 vtable 上的下标为 0
- set_color() 在 vtable 上的下标为 1
- draw() 在 vtable 上的下标为 2

原则是子类的 vtable 首先要具有和超类同样的内容。但在方法重写的情况下会替换原有的位置，而在子类中增加方法的时候，子类的 vtable 也会变得比超类的长（图 8-6）。

图 8-6 在单继承的情况下的多态



使用这个处理方式，缺点在于每次调用方法时都要引用 vtable，所以多少会有些延迟。而优点在于不论类的继承关系有多深，或者类中的方法有多多，方法调用的开销都是基本相同的。

C++ 只要在不使用多继承的时候，通常都是用这种处理方式实现继承的。但 C 语言就不可能用这个处理方式实现继承和多态了（GTK+、X-Window Toolkit 等）。

Diksam 的类只能单继承，但是可以多继承接口。因此，对于字段来说是以单继承为前提的，但是对于方法的调用来说，就不得不考虑多继承的情况。在多继承时，vtable 的下标就失去了唯一性，也就无法用这种处理方式来应对了。

8.3.3 多继承——C++

我们以 C++ 为例，看一下它是如何实现多继承的。 细节的地方因处理器而异，比如有继承了类 A 和 B 的类 C，首先 A 和 C 如果是“主要继承关系”的话，那么不论是字段还是方法，都可以使用和单继承时相同的处理方式，但问题在于 C 的对象在被当做 B 引用（将 C 向上转型为 B）的时候。C++ 在此时会将指针本身转换为 B 的 vtable 对应的地址。

这个现象可以在代码清单 8-6 中得到验证。

代码清单 8-6
C++ 的指针转换

```
#include <stdio.h>

class A {
    public:
        int a;
        virtual void a_method() {}
};

class B {
    public:
        int b;
        virtual void b_method() {}
};

// 继承了类 A 和类 B 的类 C
class C : public A, public B {
    public:
        int c;
        void a_method() {}
```

```
        void b_method() {}
};

int main(void)
{
    C *c = new C();

    // 显示 new 过的变量 c 的指针
    printf("c..%p\n", b);

    // 将其赋值到 B* 型的变量中，并显示
    B *b = c;
    printf("c as B..%p\n", b);

    // 显示 A， B， C 各类中的成员变量（字段）的地址
    printf("&A->a..%p\n", &c->a);
    printf("&B->b..%p\n", &c->b);
    printf("&C->d..%p\n", &c->c);

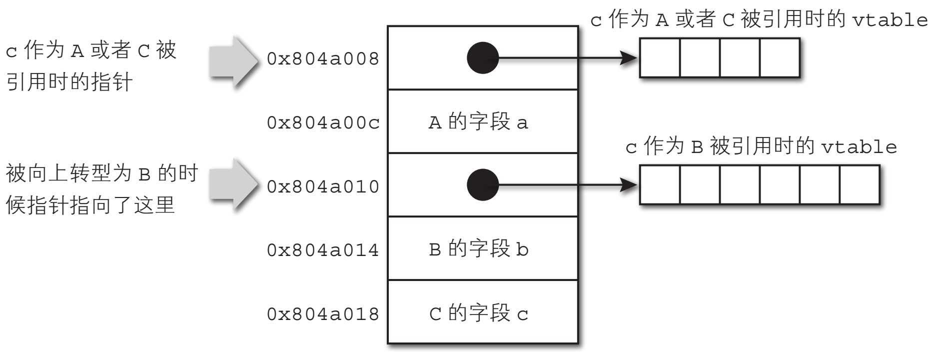
    return 0;
}
```

在我的环境中，输出结果如下（我的环境 int 和指针都是 4 个字节）。

```
c..0x804a008
c as B..0x804a010
&A->a..0x804a00c
&B->b..0x804a014
&C->d..0x804a018
```

内存结构如图 8-7 所示。在将指针赋值给 B* 型的变量时，指针所指向的地址发生了改变。可以确认在 A 的字段 a 和 B 的字段 b 之间存在着 B 的 vtable 占用的内存空间。

图 8-7
C++ 的多继承



在 C++ 中，只有这样的（自动的）转换才可能改变指针的值，C 像是通过 void* 保存了对象，给人的感觉非常不好——其实我也有过这样的经历，当然这是题外话。

如果在向上转型（通常向上转型是自动进行的）为 B 的时候移动指针，不但可以用普通的方式引用到 B 的字段，由于 vtable 也是各自保存的，因此就实现了多态。

只是在 Diksam 中，如果移动了指针的话会给 GC 的实现带来困难。现在的 Diksam，指向堆的指针类型是 DVM_Object*，但如果像 C++ 一样移动指针的话，就会由于没有指向最初的 DVM_Object 而给 GC 的标记阶段造成麻烦。另外，由于 Diksam 不能多继承字段，如果使用 C++ 的处理方式就显得有点过度设计了。

8.3.4 Diksam 的多继承

在 Diksam 中采用了如下的处理方式。

- 不是在对象的开头部分保存 vtable，而是在引用了对象的值中。这样既保存了指向对象本身的引用，同时也保存了指向 vtable 的引用。
- 在向上或者是向下转型时，替换引用值中的 vtable 就可以了。

在 Diksam 中，值被保存在 DVM_Object 联合体中，并且使用 DVM_ObjectRef 结构体引用其中的对象。

```
typedef union {
    int             int_value;      /* 值为整数时 */
    double          double_value;   /* 值为实数时 */
    DVM_ObjectRef   object;         /* 值为对象时 */
} DVM_Value;
```

DVM_ObjectRef 定义如下。

```
typedef struct {
    DVM_VTable *v_table; /* vtable 的指针 */
    DVM_Object *data;    /* 数据本身 */
} DVM_ObjectRef;
```

在 8.3.2 节中介绍了在 Diksam 中类继承的处理方式——使用 vtable 实现多态，并且在转型为接口的时候替换引用值中的 vtable（DVM_ObjectRef 的 v_table 成员）。

当然，基于上面的处理方式也会出现对象不知道自己原本是什么类型的情

况，因此在 vtable 中保存了指向对象原本类型的指针。

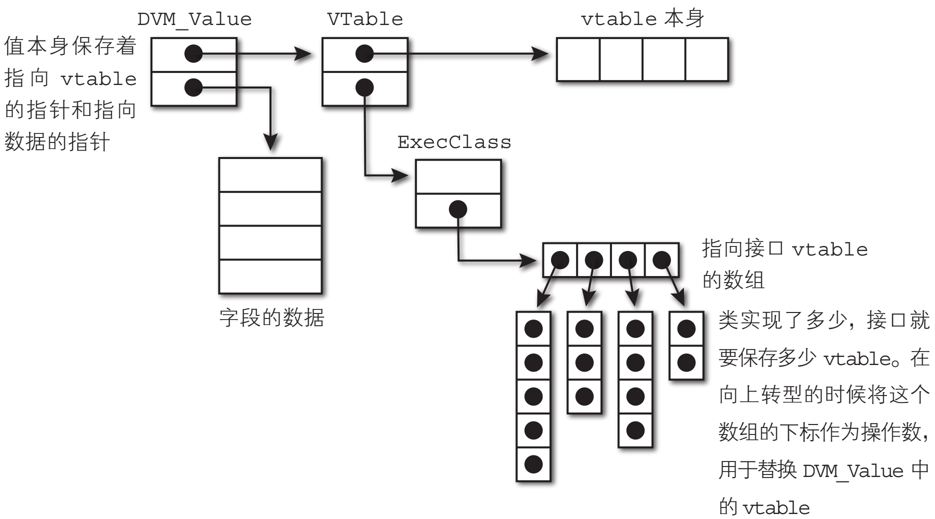
```
struct DVM_VTable_tag {
    ExecClass   *exec_class; /* 指向类的指针 */
    int         table_size;  /* vtable 的元素数 */
    VTableItem  *table;      /* vtable 本身 */
};
```

其中叫作 ExecClass 的类型就是保存类在运行时信息的类型，每个类只对应一个，在 DVM_VirtualMachine 中以数组的方式保存（这个数组的创建时机请参考 8.4.7 节）。

```
typedef struct ExecClass_tag {
    DVM_Class              *dvm_class;
    ExecutableEntry        *executable;
    char                   *package_name;
    char                   *name;
    DVM_Boolean            is_implemented;
    int                    class_index;
    struct ExecClass_tag   *super_class;
    DVM_VTable             *class_table;
    int                    interface_count;  ←请注意这个地方
    struct ExecClass_tag   **interface;      ←和这个地方
    DVM_VTable             **interface_v_table;
    int                    field_count;
    DVM_TypeSpecifier      **field_type;
} ExecClass;
```

如图 8-8 所示（在图中省略了接口的 ExecClass）。

图 8-8 Diksam 的多继承



如此一来，在向上转型的时候，要找到相应的 vtable 并替换 `DVM_Value` 中的（`DVM_ObjectRef` 的）`v_table` 成员。在使用 `up_cast` 指令（将在后面的章节介绍）时，这个接口的数组下标将会作为操作数进行传递。

补充知识　无类型语言中的继承

作为像 Diksam 和 C++ 这样的静态类型的语言，不论是引用字段的时候还是调用方法的时候，都能够一下子引用到在编译时 * 就已经决定的索引值。多继承的情况则更加复杂，但不论怎样都可以使用“固定的开销”引用到字段或者方法，这点是没有变化的。

* 由于 Java 的 class 文件中成员名字是字符串，因此发生在加载 / 链接时。

与此相对，在像 Ruby 这样没有变量类型的语言中，`a.hoge` 语句（单纯的实现）由于在编译时并不知道 a 的类型，没有办法一下子就访问到索引值，因此必须要利用成员名字进行搜索。也就是说，根据成员数量不同，检索需要的时间也不同，所以引用成员时就不是“固定的开销”了。

尽管如此，如果使用二分法检索（假如有 1000 个成员的话，用 10 次以内的循环就可以完成）的话，也可以看作是“固定的开销”。

8.3.5 重写的条件

重写是在调用超类的方法时，实际被调用的是（也许是）子类的方法。从调用者看上去好像调用的是超类的方法，但是实际上调用的却是子类的方法，而这个子类的方法说不定会让调用者吓一跳。

例如，子类方法的返回值类型要比超类的“窄”（被称为**共变**，covariant）。当超类有以下方法时，如果子类要对其进行重写的话，子类的 `getShape()` 方法不能返回 Shape 以外的类型 *。

* 当然，`null` 是个例外。

```
Shape getShape();
```

子类的返回类型只要是比超类“窄”就算合法。也就是说，像下面这样将 `Shape` 的子类作为返回值的重写是合法的。

```
// Circle 如果是 Shape 的子类的话，这个方法就是合法的
Circle getShape();
```

这种情况下，子类的 `getShape()` 方法一定只能返回 Circle 类型。但是在调用者，期待的是比 Circle 更“宽”的 `Shape` 类型，这是没有问题的。相反，如果期待的是 `Circle`，返回的却是其他 `Shape`（`Line` 或者 `Rectangle`）的话，真的

会被吓一跳吧。

像这样将返回值共变的好处有很多，比如限制了 Java 中 Object 类的 clone() 方法的返回值，以减少不必要的转型。Java 从 JDK1.5 开始具备这个功能。

在 Diksam 中，参数的情况却是相反的。子类方法的参数类型要比超类的“宽”（被称为**反变**，contravariant）。在超类中有如下方法时：

```
void drawShape(Circle circle);
```

在子类中可以合法地定义如下方法：

```
void drawShape(Shape shape);
```

子类的 drawShape() 能接受的参数，超类的 drawShape() 应该也能接受，这样的话调用者就不会被吓一跳了。

至于访问修饰符，子类的方法必须要比超类的“宽松”。public 的方法不能被重写为 private 的，反之则合法。

9.2.3 节会为 Diksam 引入异常的概念，Diksam 的异常处理属于 Java 风格的异常检查（把方法中可能抛出的异常全部用 throws 列出，并由编译器检查）。这种情况下，子类的方法不能比超类抛出更多的异常。

让我们再说回参数，Diksam 在编译器进行静态检查时允许反变是非常方便的设计，但是在实际应用中，有时也需要同时允许共变的存在。假设 Shape 有设置样式的 setStyle(Style style) 方法，Line 和 Circle 需要的样式也应该是根据不同图形定制的 Style 的子类——但是，如果实现了这项功能的话，一定要在调用 Shape 的 setStyle() 时进行一些运行时检查。

相似的话题在 7.3.2 节的补充知识中已经作过介绍，在 Java 中假设存在超类 Shape 和子类 Circle，那么 Circle 的数组会自动成为 Shape 数组的子类。这个设计可能会引发叫作 ArrayStoreException 的运行时错误。

如果将 Shape[] 和 Circle[] 看作是类的话，就应该能看出来它们是共变参数的一种。

```
// 将“Shape 数组”看作是类
class ShapeArray {
    // 取得数组元素的方法
    Shape get(int index);
    // 设置数组元素的方法
    void set(int index, Shape shape);
```

```
}

// CircleArray 是 ShapeArray 的子类
class CircleArray extends ShapeArray {
    // 返回值的共变 → 没问题
    Circle get(int index);
    // 参数类型的共变 → 必须进行运行时检查
    void set(int index, Circle circle);
}
```

8.4　关于类的实现

本节将要介绍的是在类的实现中除了继承和多态的部分。

8.4.1　语法规则

Diksam 在声明变量的时候和 Java 相同，形式如下所述：

```
类型 变量名 ;
```

我们在 6.1.5 节中曾提到，有些语言是将类型写在后面的。

```
// 以 ActionScript 为例
var a:int;
```

将类型写在后面的语法结构在制作语法分析器时会非常轻松，但是在 Diksam 中，使用了 C 和 Java 程序员习惯的语法结构，即将类型写在前面——这么做了才知道，这是一条坎坷的路。

如果以“类型 变量名；”的语法进行声明，在一开始就能得到“类型”，从而一下子就可以知道这是一条声明语句，但也只有 `int` 和 `double` 可以作为保留字，`Point` 之类的类名字就不能够成为保留字了。因此，只看这个是不能知道类名字的。yacc 虽然会预读一个符号，但是，例如下面这样的数组声明，

```
Point[] p;
```

即使是预读了 `Point` 后面“`[`”也不能搞清以下两点：

- 是数组型变量 `Point` 要通过 `[]` 引用元素？
- 还是要声明 `Point` 类的数组？

在 C 中，要想使用类型 Point，就必须要在使用的地方前面声明这个类型。在这种语言中使用的处理方式是：在声明的时候把类型由解析器传递给词法分析器，词法分析器随后把它登记为标识符，之后的 Point 就会被当做类型名称来处理。但是，作为一门当下的编程语言来说似乎不太雅致。在 C 语言中，为了表现诸如 `Husband` 引用了 `Wife`，`Wife` 又引用了 `Husband` 这样的相互引用，不得不使用“预先声明结构体标签”的怪异方法。

因此，Diksam 的语法规则如下：

```
declaration_statement
        : type_specifier IDENTIFIER SEMICOLON ←省略了初始化等操作
        ;

type_specifier
        : basic_type_specifier ←表示 int、boule 等基本类型
        | array_type_specifier
        | class_type_specifier ←表示类名
        ;

array_type_specifier
        : basic_type_specifier LB RB
        | IDENTIFIER LB RB
        | array_type_specifier LB RB
        ;

class_type_specifier
        : IDENTIFIER
        ;
```

看了上面这段代码，可能有人会想了：

> 等等。刚才不是说即使预读了 `Point` 后面的一个符号也搞不清楚它到底是变量名还是类名吗？但是在这个规则中不是很明显地告诉你如果只有一个 `IDENTIFIER` 的话就是 `class_type_specifier` 吗？

这个语法规则的重点在于引入了 `array_type_specifier`。如 7.2.1 节中所述，在引入类之前，数组声明语法如下：

```
type_specifier
        : basic_type_specifier
        | type_specifier LB RB
        ;
```

在引入了类之后，我认为与 `basic_type_specifier` 一起处理会更好。

```
type_specifier
        : basic_type_specifier
        | IDENTIFIER             ←增加了这行代码
        | type_specifier LB RB
        ;
```

但是，还是像前面说的，在预读了 IDENTIFIER 后的“[”后，yacc 就会因为不知道是把它当做要引用数组元素继续 shift 好，还是当做 type_specifier 进行 reduce 好而发生错误。顺带提一下，引用数组元素的规则如下：

```
primary_no_new_array
        : primary_no_new_array LB expression RB
        | IDENTIFIER LB expression RB
        ;
```

不过，引入了 array_type_specifier 后，即使预读到了“[”也不会进行归约，因而也不会进行归约 / 归约冲突和移进 / 归约冲突。当然，现在还是搞不清楚到底是在 array_type_specifier 中，还是在 primary_no_new_array 中。在看到了代码清单 2-5 的 y.output 的后半部分中一并记载了多个规则就能明白，yacc 的状态可以跨越多个规则，并不会引发问题。

8.4.2　编译时的数据结构

函数在编译时被保存在 FunctionDefinition 结构体中，然后被复制到 DVM_Executable 中的 DVM_Function 结构体。类保存在编译时的数据类型 ClassDefinition 和 DVM_Executable 的 DVM_Class 中。

对于从开头一直读到这里的读者，我想就没有必要太过详细地介绍了，还是一笔带过吧。

首先是编译时的数据结构 ClassDefinition。

```
struct ClassDefinition_tag {
    DVM_Boolean is_abstract;
    DVM_AccessModifier access_modifier;
    DVM_ClassOrInterface class_or_interface;
    PackageName *package_name;
    char *name;
    ExtendsList *extends;       ←fix 之前的临时数据结构
    ClassDefinition *super_class;
    ExtendsList *interface_list;
```

```
        MemberDeclaration *member;   ←成员
        int line_number;
        struct ClassDefinition_tag * next;
    };
```

这个结构体中值得注意的地方是，成员 extends 是一个在 create.c 中被构建后，又在 fix_tree.c 中被抛弃的临时数据结构。在 Diksam 中，继承类和接口时，没有使用 Java 的 extends 和 implements 风格，而是使用了 C++ 和 C# 的冒号，因此（一开始）无法区分类和接口。这里先姑且加入一个 extends 成员，之后将在 fix_tree.c 中分为 super_class 和 interface_list（fix_extends() 函数）。

member 保存了类的成员。类成员的字段和方法以联合体的形式被保存在 MemberDeclaration 结构体中。

```
    struct MemberDeclaration_tag {
        MemberKind kind;
        DVM_AccessModifier access_modifier;
        union {
            MethodMember method;
            FieldMember field;
        } u;
        int line_number;
        struct MemberDeclaration_tag *next;
    };
```

先看一下字段的定义。

```
    typedef struct {
        char             *name;
        TypeSpecifier    *type;
        int              field_index;
    } FieldMember;
```

name 和 type 表示字段的名称和类型。

这里要稍稍介绍一下 field_index。在 DVM 中，各个对象的字段的数据都以 DVM_Value 数组的形式保存。这里的 field_index 就是这个数组的下标，在 fix_tree.c 中进行设置。

继承类的时候，超类的字段会由子类继续持有（如图 8-5）。MemberDeclaration 结构体的列表只含有当前类的成员，并不包含超类的成员，但 field_index 却和超类的字段含有的通用的编号。

在 Diksam 中就是像这样，在编译时指定字段索引值。这是因为 Diksam 中并

没有相当于 Java 的 class 文件这样的产出物。如果字节码保存在文件中的话，类中的字段增加，索引值也可以在文件中随之增加，但是却做不到上面的方法。在 Java 的字节码中指定的并不是字段的索引值，而是字段的名称。

接下来是方法。

```
typedef struct {
    DVM_Boolean         is_constructor;
    DVM_Boolean         is_abstract;
    DVM_Boolean         is_virtual;
    DVM_Boolean         is_override;
    FunctionDefinition  *function_definition;
    int                 method_index;
} MethodMember;
```

首先 method_index 和 field_index 一样，是一个包含了超类方法的通用编号。但是，在实现了接口方法的时候，这个方法的 method_index 只在接口之间通用。换句话说，它就是 vtable 的下标（如图 8-8）。

另外，如代码所见，MethodMember 中包含了指向 FunctionDefinition 的指针。就是说，对于 Diksam 来说，方法也不过就是函数（稍有不同），它被注册在 FunctionDefinition 中的同时也会创建 DVM_Function。

FunctionDefinition 中增加了从方法能够追溯到类的 ClassDefinition 的指针。如果这个成员为 NULL 的话，就说明它不是方法而只是普通的函数。

```
struct FunctionDefinition_tag {
    ···前面省略···
    ClassDefinition *class_definition; ←指向类的指针
    ···后面省略···
};
```

8.4.3 DVM_Executable 中的数据结构

编译完成后就要生成 DVM_Executable 了，这里使用了 DVM_Class 结构体来保存 DVM_Executable 中的类。

DVM_Class 在 DVM_Executable 中以下面这种可变长数组的形式保存。

```
struct DVM_Executable_tag {
    ··· 前面省略 ···
    int         class_count;
```

```
    DVM_Class   *class_definition;
    ··· 后面省略 ···
};
```

DVM_Class 结构体的定义如下所示。

```
typedef struct {
    DVM_Boolean              is_abstract;
    DVM_AccessModifier       access_modifier;
    DVM_ClassOrInterface     class_or_interface;
    char                     *package_name;
    char                     *name;
    DVM_Boolean              is_implemented;
    DVM_ClassIdentifier      *super_class;
    int                      interface_count;
    DVM_ClassIdentifier      *interface_;
    int                      field_count;
    DVM_Field                *field;
    int                      method_count;
    DVM_Method               *method;
} DVM_Class;
```

上面的代码中出现了 DVM_ClassIdentifier，它是由包名和类名组成的类型。这个类型可以保存超类和（实现了的）接口。

字段和方法都是以可变长数组的方式保存的。DVM_Class 中保存的只有当前类中的定义，并不包含超类的部分。

保存字段的 DVM_Field 的定义如下。其中成员所表示的含义都显而易见。

```
typedef struct {
    DVM_AccessModifier      access_modifier;
    char                    *name;
    DVM_TypeSpecifier       *type;
} DVM_Field;
```

方法也是一样。

```
typedef struct {
    DVM_AccessModifier  access_modifier;
    DVM_Boolean         is_abstract;
    DVM_Boolean         is_virtual;
    DVM_Boolean         is_override;
    char                *name;
} DVM_Method;
```

并且，方法“差不多”是普通的函数，只是在动作上有细微的不同，因此在 DVM_Function 中保存了它是不是方法的标识符（下面的 is_method）。

```
typedef struct {
    DVM_TypeSpecifier   *type;
    char                *package_name;
    char                *name;
    int                 parameter_count;
    DVM_LocalVariable   *parameter;
    DVM_Boolean         is_implemented;
    DVM_Boolean         is_method;  ←增加了这个成员
    ···后面省略···
} DVM_Function;
```

方法的函数名以**类名#方法名**的方式保存在 DVM_Function 的 name 成员中。例如 Point 类的 print() 方法，在 DVM_Executable 的时候也可以看作是名称为 Point#print、is_method 为 true 的普通函数。当然，解析器是不允许函数名中包含 # 的，因此，使用者即使自己写了名为 Point#print() 的函数，也不能在自己的程序中通过 Point#print() 的方式调用。

8.4.4 与类有关的指令

随着引入了类的概念，DVM 中也增加了相关的指令，如表 8-2 所示。下面的 object 型为字符串、数组、类对象的总称（在表 7-1 中为字符串和数组的总称，这次增加了类的对象）。

表 8-2
随着引入类而增加的指令

指令	操作数类型	含义	栈动作
push_field_int	short	根据操作数（即索引值）取得栈顶对象的属性（int 型）值，并将其入栈	[object]→[int]
push_field_double	short	根据操作数（即索引值）取得栈顶对象的属性（double 型）值，并将其入栈	[object]→[double]
push_field_object	short	根据操作数（即索引值）取得栈顶对象的属性（object 型）值，并将其入栈	[object]→[object]
pop_field_int	short	用栈顶的值（int）赋值给栈顶的对象中与操作数（即索引值）对应的属性	[int object]→[]

（续）

指令	操作数类型	含义	栈动作
pop_field_double	short	用栈顶的值（double）赋值给栈顶的对象中与操作数（即索引值）对应的属性	[double object]→[]
pop_field_object	short	用栈顶的值（object1）给栈顶的对象中与操作数（object2）对应的属性赋值	[object1 object2] →[]
push_method	short	根据操作数（即索引值）取得栈顶对象的方法的索引值，并将其入栈（请参考 8.4.5 节）	[object]→[object int]
new	short	创建与操作数（即索引值）对应的类的实例，并将它的引用入栈。此时并不会调用构造方法，因此接下来会另外创建构造方法的调用代码	[]→[object]
up_cast	short	将栈中的对象引用的 vtable 替换为操作数指定的接口（请参考 8.3.4 节）	[object]→[object]
down_cast	short	检查栈上的对象引用是否能向下转型，并在此基础上将它的 vtable 替换为操作数指定的接口的 vtable（请参考 8.4.9 节）	[object]→[object]
duplicate_offset	short	从栈顶取得操作数指定个数的元素，将其复制并入栈	[]→[object]
super		将栈顶的对象引用的 vtable 替换成其父类的	[object]→[object]
instanceof	short	返回栈顶的对象是否属于操作数（即索引值）指定的类型	[object]→[boolean]

访问字段的指令，把字段的索引值作为操作数传递就可以了。这里的字段值指的是保存在 FieldMember 结构体中的 field_index。

除此之外的指令将在后面的章节中介绍。

补充知识　方法调用、括号和方法指针

在 Diksam 中调用方法或函数的时候，会像 p.get_x() 这样使用括号。即使方法中一个参数都没有，括号也不能省略。

* 在 Ruby 中即使有参数也可以省略括号。

但是，根据语言的不同，有的括号也是可以省略的。例如在 Eiffel 的语言中一个参数都没有的时候就可以省略掉括号 *。这种做法的优点只是单纯地改善了外观问题，节约了录入量。

在 Java 和 C++ 中，为了实现封装，普遍的做法是将字段设置为 private，然后再像下面这样编写访问器（accessor）。

```
public double get_x() {
    return x;
}
```

这里将来可能会发生变化，例如也许不会把 x 单单作为字段保存，而是将计算后的结果返回去。考虑到这种情况，创建访问器的方法比起把字段设置为 public 公开出去要好。但是，编写访问器是一件非常麻烦的事。

在这点上 Eiffel 的做法是，最初的字段是公开的，如果中途想要修改为在计算后再返回结果的话，此时可以定义一个名字为 x 的方法替换掉最初公开的字段。不论是字段还是方法，从使用者的角度来看都是 p.x，这种做法达到了在不影响使用者的前提下将字段替换为方法的目的。

说起替换 x，在 Eiffel 中默认是不能使用 p.x 为其赋值（使用“.”对字段进行引用时不能作为左值）。因此，每个字段的访问器都是在一开始就强制创建的。我很同意这种做法，从外部改变字段的值是件大事，因此必须编写方法来实现[①]。

我认为这是一个不错的主意，但在 Diksam 中并没有使用。原因是，采用了这种方式的话方法本身就不能再作为值被处理了。

```
// set_on_click方法中传递了对象o的方法作为参数。
button.set_on_click(o.method);
```

上面的代码注册了一个事件，在按下 GUI 的按钮时相当于调用了 o.method()。设想一下，如果把方法指针用作事件句柄或者回调方法的话，会发现调用 o.method 方法是件困难的事情（因为要向 set_on_click 传递 o.method() 的执行结果）。

将方法自身作为值处理的功能，将在 9.5.4 节中实现。

8.4.5　方法调用

Diksam 中 push_method 指令用于在考虑了多态的情况下，决定被调用的方法。

push_method 与 push_function 把函数入栈的功能相似，是把方法入栈。实际上被推入栈中的也和函数一样，是 DVM_VirtualMachine 中 Function 类

① 意思是让编程人员知道自己开放了哪些字段是可以从外部赋值的。——译者注

型的对应表的索引值。作为操作数的索引值，跟字段的情况差不多，是 MethodMember 结构体的 method_index。push_method 会根据栈中的对象和 method_index，在考虑到多态的情况下选择适当的函数。

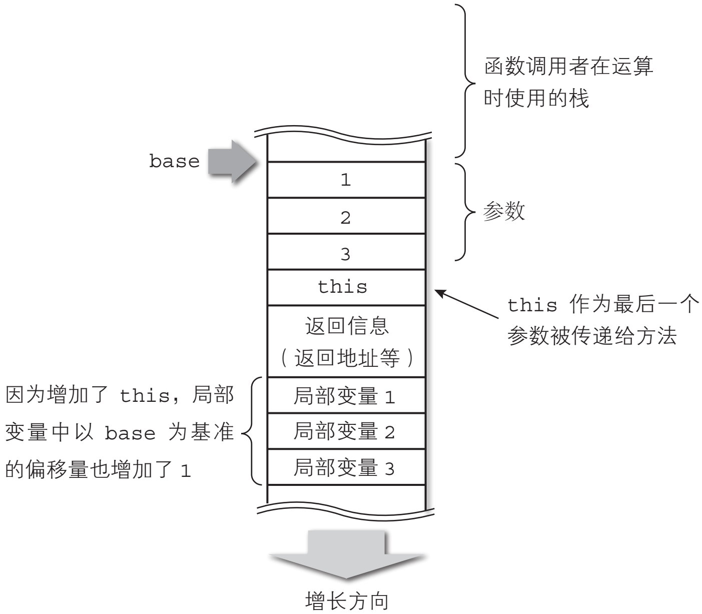
push_method 在选择了适当的方法之后就要进行调用了，这个动作与调用普通的函数基本相同，使用的指令也都是 invoke。

Diksam 的函数调用已经在 6.4.3 节中介绍过了。

但是，在调用方法的时候，必须要将目标对象传递给方法。在被调用的方法中这个对象将被作为 this 进行引用。

在 Diksam 中 this 作为最后一个参数传递给方法。如图 8-9 所示。

图 8-9
方法调用



在 Diksam 中，参数按照从前往后的顺序入栈，此时 this 是最后一个参数，成为了栈的顶端。因此，push_method 的时候可以引用 this 选择方法。

另外，如图 8-9 中所示。局部变量的偏移量也因为增加了 this 而增加了 1。这个调整在加载时进行（请参考 load.c 的 convert_code() 函数）。

在 new 时调用的构造方法多少会有些不同，操作步骤如下：

1. 首先，创建对象并将其引用入栈。
2. 将构造方法的参数按照从前到后的顺序入栈。
3. 使用duplicate_index指令将1.中创建的对象引用复制到栈的顶端。

4. 使用invoke指令调用方法。
5. 最后，在栈中只留下了1.中入栈的对象引用。

8.4.6 super

Diksam 和 Java 一样，有 super 关键字。使用它可以调用到 this 的超类的方法。

在 Diksam 中 super 的实现非常简单，只将 this 指向的 DVM_ObjectRef 的 vtable 替换成超类即可。

但是，并没有把让 super 保存在其他变量中的使用方法考虑在内（如“p = super;”）。因此，除了 super. 方法名 () 以外的形式，其他在编译时都会报错。

并且，在 Diksam 中超类的构造方法并不能通过 super() 的形式调用，而是必须要使用 super.initialize() 的形式。在 Diksam 中，不显式地指定构造方法名称的话，就不能知道调用的是哪个构造方法。

8.4.7 类的链接

类与函数相同，也会引用多个文件，因此必须要进行链接。

这里的结构和函数的基本相同。下面进行简单地介绍。

在 8.4.3 节中提到过，DVM_Executable 中保存着 DVM_Class 的数组。new 指令以操作数的形式保存着类的索引值，这个索引值就是 DVM_Executable 中 DVM_Class 的下标。因此，即使没有在当前代码中定义，只是单纯被使用到的类也会被注册到 DVM_Class 数组中，而且这样的类，它的 is_implemented 为 false（请参考 8.4.3 节）。

将 DVM_Executable 加载到 DVM 的时候，push_function 的操作数将会替换为 DVM 内的下标（请参考 6.4.1 节）。类的话，new 的操作数将会被替换为 DVM 内的下标（load.c 的 convert_code() 函数）。

这里说的“DVM 内的下标”是指 DVM_VirtualMachine 中 ExecClass 数组的下标。ExecClass 创建于加载包含类的源文件的时候，同时也会扩展 ExecClass 数组（load.c 的 add_classed() 函数）。

is_implemented 为 false 的 DVM_Class 会在此时嵌入方法的实现。

8.4.8 实现数组和字符串的方法

如同在 8.2.7 节中写到的，Diksam 的数组和字符串有不少内建方法。

不论是数组还是字符串在创建对象的时候，都在引用对象的 DVM_ObjectRef 中以保存硬编码 vtable 的方式实现。在这个 vtable 中注册了数组或者字符串方法的（原生方法）实现。

这样就可以在运行时调用方法了，但在编译时必须要进行参数检查。最麻烦的是，例如确定数组的 insert() 方法中第 2 个参数（要插入的数组元素）的类型。int 数组第二个参数必须是 int，Point 类的数组则必须是 Point。

认真考虑这个问题的话，就会想到 Java 的泛型（Generics）和 C++ 的模板这些功能。但是，现在没有必要只是为了数组使用这么复杂的处理方式，用下面的方式也可以解决（保存一个包含了内建方法参数的类型信息的数组）（fix_tree.c）。

```
/* 参数的类型和数量保存在 static 的数组中
 * DVM_BASE_TYPE 表示数组元素的类型。
 */
static DVM_BasicType st_array_size_arg[] = {};
static DVM_BasicType st_array_resize_arg[] = {DVM_INT_TYPE};
static DVM_BasicType st_array_insert_arg[] = {DVM_INT_TYPE, DVM_BASE_TYPE};
static DVM_BasicType st_array_remove_arg[] = {DVM_INT_TYPE};
```

虽然我觉得这么做不太优雅，但还是在 DVM_BasicType 中加入了奇怪的元素（DVM_BASE_TYPE）。

8.4.9 类型检查和向下转型

类的类型检查（instanceof）和向下转型其实是两个相似的功能。向下转型在执行时也会进行和 instanceof 相同的类型检查。

因此，instanceof 和向下转型可以看作是编译时进行的检查。首先，Diksam 中存在着类和接口，在 A instanceof B 的时候，会出现以下几种情况。

A	B	
类	类	只有在 A 和 B 为同一类，或 A 是 B 的超类时，才有可能返回真。
接口	类	只有在 B 实现了 A 的情况下，才有可能返回真。
类	接口	在 A 的子孙中只要有实现了 B 的，通常会返回真。
接口	接口	对象实现了 A 和 B 两个接口的时候，通常会返回真。

也就是说，instanceof 的右边指定了类的时候，编译时会因为绝对不会为真的 instanceof 而导致错误。向下转型时也是一样，如果类之间没有继承关系的话也不可能进行向下转型。

更为重要的是，在 Diksam 中，绝对为真的 instanceof（同类之间的和与超类进行的 instanceof）、向超类的转型也会导致编译错误。我认为，没用的代码最后会导致 bug，在编译时应该阻止这种情况发生。

运行时的检查，将在 ExecClass 结构体中遍历所有超类和接口（因此，速度不是很快）。

向下转型的时候，在进行了和 instanceof 同样的检查后，如果转型目标是类，就将引用中保存着的 vtable 替换为目标类的 vtable。如果转型目标是接口的话，就用目标接口的 vtable 替换。

补充知识　对象终结器（finalizer）和析构函数（destructor）

在 Java、C++、C# 等语言中，类对象销毁时可以通过用户程序得知。具体来说，在对象销毁时，Java 会使用终结器，C++ 和 C# 则会调用被称为析构函数的方法。

但是，Diksam 中没有这个功能。姑且不论必须完全由编程人员控制对象寿命的 C++，和可以预测对象销毁时机的 Python（使用基本的引用计数器类型 GC，在不创建循环引用的前提下），Diksam 这样的语言中即使创建了对象终结器 * 也没有什么作用。

对象终结器的用途，比如说用来关闭在 C 中 fopen() 返回文件的指针。因为能够打开的文件数（译注：文件句柄）在进程中是有上限的，所以使用 fopen() 打开的文件在用完后应该立即关闭。但是这个处理如果要交给对象终结器的话，它也不知道要何时进行哪些操作（特别是 Java 中连有没有进行动作都不知道）。说不定在处理开始前，所有的文件句柄就已经用光了。所以说这种做法不保险。

如果像上面说的那样，我想还是不要定义对象终结器。如果能够简单地实现对象终结器的话，也许效果会更好。但是，实际上想要优雅地实现对象终结器并非易事。

也许有人会想，"嗯？释放对象空间之前，不是只会调用 finalize() 方法吗？"如果在对象终结器中把 this 赋值给了全局变量或者其他东西的话怎么办？

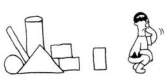
垃圾回收器会根据"不能被追溯到的对象"这一原则来判断对象是否不被需要。但

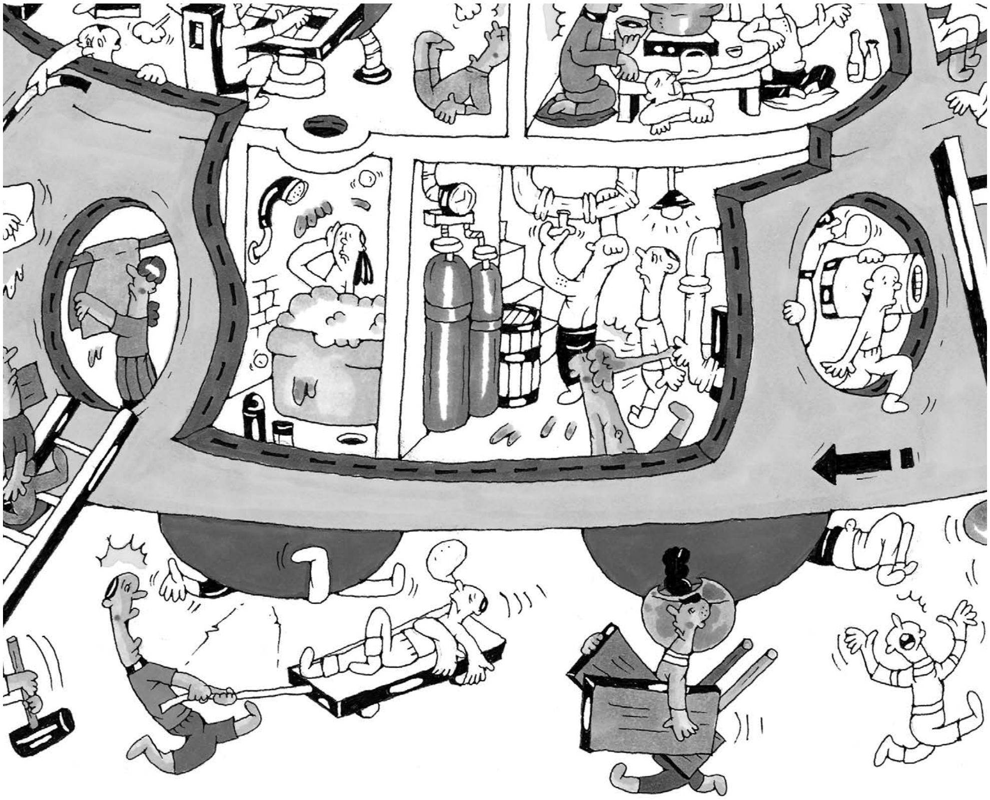
* 这里采用了 Java 的说法。

是，在对象终结器中将 this 赋值给了全局变量的话，此时这个对象就可以从全局变量中被追溯到，以至于不能被释放了。

Java 的 GC 也不得不面对这个问题。目前已知的是如果使用了对象，终结器会使 GC 的效率大幅下降。

另外，crowbar 和 Diksam（book_ver.0.4）中，对于指向原生指针类型的对象来说，可以使用原生函数实现对象终结器。编写原生函数多少会包含一些危险的处理，因此不得不考虑到，如果对在原生函数中悄悄地把对象释放了，然后又被其他程序引用的话就会导致崩溃，那么只能后果自负了。





第 9 章

应用篇



9.1 为 crowbar 引入对象和闭包

在第 8 章中为 Diksam 引入了类，但是目前的 crowbar 还不支持面向对象功能。

而在当今的编程语言中，大有如果不支持面向对象就不会被认可的架势，因此也要让 crowbar 支持面向对象。

但是，crowbar 的面向对象与 Diksam、C++、Java、C# 等相比会有一些不同。它没有类的概念。

本节将要介绍的，就是 crowbar 的面向对象具体是怎样设计和实现的。

9.1.1 crowbar 的对象

像前面提到的，crowbar 中没有类的概念。因此，创建对象的时候也不需要指定类和使用 new 指令，只是单纯地调用原生函数 new_object() 即可。

```
o = new_object();
```

crowbar 中的对象与 C 对象的结构体相同，也可以保存**成员**。但是，由于没有类声明也没有结构体声明，因此成员将在运行时以赋值的方式增加。

```
p = new_object();
p.x = 10;
p.y = 20;
print("p..(" + p.x + "," + p.y + ")\n");
```

肯定有人会说（我必须承认我也是其中一员）：“没有类型声明感觉真是别扭。”但起码这个方式实现了类似 C 的结构体的功能。

并且，与 Diksam、Java 一样，这个对象也是引用类型的，使用 new_object() 返回了一个指向对象的引用。因此，下面的代码将输出 20。

```
o1 = new_object();
o1.hoge = 10;
o2 = o1;
o2.hoge = 20;
print(o1.hoge);
```

上面这段代码的内存映像如图 9-1 所示。

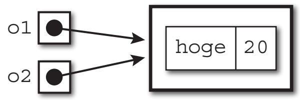


图 9-1
crowbar 的对象是引用类型

9.1.2 对象实现

首先，关于“对象”这个词无论是在 crowbar 中还是在 Diksam 中，都是指“在堆中被创建的”字符串、数组、类的“对象”。像 crowbar 的 `CRB_Object`、Diksam 的 `DVM_Object` 结构体，这些都是通过联合体保存的“在堆中被创建的对象”。

在 Diksam 中，堆中类的实例的对象被称为“类对象”或者“类的实例”。但是，crowbar 中并没有类的概念。

对于使用者来说，应该把数组叫作数组，把字符串叫作字符串才对吧。因此，`new_object()` 函数返回的也应该叫作“对象”。但是，从 crowbar 的实现上来看，由于已经引入了叫作 `CRB_Object` 的结构体，因此不得不给 `new_object()` 返回的东西起一个其他的名字。

这里把这个东西叫作 assoc。assoc 是**关联数组**（associative array）的简称。

之所以被称为关联数组（最近不知道为什么有很多语言叫它“哈希”*），是因为它是可以以字符串（等）为键从中取出值来的数组。正是由于可以以字符串为键取值这点，crowbar 的对象最终使用了关联数组的方式 *。

* 在 Perl 的 ver.4 中还是叫作“关联数组”，但从 ver.5 开始就叫作“哈希”了。我认为哈希只不过是一个实现方式，因此还是“关联数组”的名字更为贴切。

* 现在作为键的字符串还不能使用变量，因此还不能作为关联数组投入使用。顺便提一下，JavaScript 中可以使用 [] 访问数组元素，因此可以作为关联数组使用。

assoc 的结构体定义如下（crowbar.h）。

```
typedef struct {
    char        *name;
    CRB_Value   value;
    CRB_Boolean is_final; ←现阶段不需要在意它。
} AssocMember;

struct CRB_Assoc_tag {
    int         member_count;
    AssocMember *member;
};
```

assoc 的成员是名称和值的组合，因此 assoc 中只保存了 `AssocMember` 的可变长数组。`AssocMember` 中的 `is_final` 是个谜一样的成员，因为它是一个不能赋值的局部变量，所以现阶段不需要在意它。

在现在的实现中，每次增加成员都会使用 `realloc()` 来扩展可变长数组，并且以线性的方式搜索。当然它的速度不快，这时候只要拿出富翁式编程的免死金牌来就好了。

因为要引用 assoc，所以要修改 `CRB_Object`。

```
typedef enum {
    ARRAY_OBJECT = 1,
    STRING_OBJECT,
    ASSOC_OBJECT,          ←增加了这行代码
    SCOPE_CHAIN_OBJECT,    ←将在后面说明
    NATIVE_POINTER_OBJECT,
    OBJECT_TYPE_COUNT_PLUS_1
} ObjectType;

struct CRB_Object_tag {
    ObjectType type;
    unsigned int        marked:1;
    union {
        CRB_Array       array;
        CRB_String      string;
        CRB_Assoc       assoc;          ←增加了这行代码
        ScopeChain      scope_chain;    ←将在后面说明
        NativePointer   native_pointer;
    } u;
    struct CRB_Object_tag *prev;
    struct CRB_Object_tag *next;
};
```

与此同时还增加了 `ScopeChain` 和 `NativePointer` 这两个怪东西，还是留在后面介绍吧。

9.1.3 闭包

对于对象来说不能够只有数据成员，还得有方法吧？我好像听见有人问我："方法怎么着了？"但是，我还是要把这些急脾气的人放在一边，先讨论一下别的话题。

在 crowbar 中可以使用**闭包**（closure）功能。所谓闭包，就是可以在表达式中定义函数。

```
# 创建闭包
c = closure(a) {
```

```
    print("a.." + a);
};

# 调用闭包
c(10);
```

closure 是创建闭包的关键字。通过在后面的括号内定义形式参数、在程序块中编写代码来创建闭包。上面的代码中把创建的闭包赋值给了变量 c。

利用代码 c(10) 可以调用闭包。因此，这段代码会输出 a..10。

C 语言的程序员看了这个可能会想，“什么嘛！这不就是函数指针吗?”（当然，闭包可以在表达式中任意编写，比函数指针容易上手）。闭包确实有与函数指针相似的一面，实际上使用方法也是一样的。

但是，决定性的区别在于，闭包可以引用到闭包声明所在位置的局部变量。

举一个关于 foreach 的例子。在 crowbar 中循环数组的全部元素时需要编写如下代码：

```
for (i = 0; i < array.size(); i++) {
    # 处理
}
```

这种编码方式依赖于数组概念的实现。如果此时改变设计思路，不用数组而改用链表了，那么与之相关的所有地方都要进行修改。这是一件很烦人的事，因此在 C# 中出现了一个叫作 foreach 的语法。

```
foreach (Object o in hogeCollection)
{
    // 处理
}
```

Java 从 J2SE5.0 开始也增加了同样的语法。 有了这个语法确实方便了不少，虽说方便了但是还是要考虑如何把它调整为语法规则。

但是，在可以使用闭包的语言中，是可以把代码写成下面这样的 *。

* 但是，在 crowbar 的标准中，并不是引入 foreach 函数，而是引入 foreach 语法。

```
foreach(hoge_collection, closure(o){
    # 处理
});
```

这里的 foreach 并不是关键字，而是单纯程序库的函数。第 1 个参数是集合对象，第 2 个参数可以接收闭包。foreach 函数将第 1 个参数（集合对象）中的元素依次取出，并以此为参数调用第 2 个参数（闭包）。

如果只是满足调用 foreach 函数时的这个功能的话，那么 C 的函数指针也



是可以实现的。但是，在通过这种方式使用闭包的时候，如果可以从循环的内部引用外部的变量的话，就是一件不寻常的事情了。

```
fp = fopen("hoge.txt", "w");
foreach(hoge_collection, closure(o) {
    fputs(o.name, fp); # 引用循环外部的变量 fp
});
```

闭包使这种调用方式成为可能。这就是闭包与 C 语言的函数指针之间决定性的不同。

9.1.4 方法

对象和闭包组合起来，就变成了下面这段代码（代码清单 9-1）。

代码清单 9-1
Point.crb（之一）

```
# 创建“点”的函数（构件方法）
function create_point(x, y) {
    this = new_object();
    this.x = x;
    this.y = y;

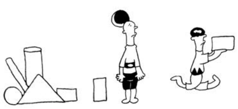
    # 定义输出坐标的方法 print()
    this.print = closure() {
        print("(" + this.x + ", " + this.y + ")\n");
    };
    # 定义移动坐标的方法 move()
    this.move = closure(x_vec, y_vec) {
        this.x = this.x + x_vec;
        this.y = this.y + y_vec;
    };
    return this;
}

# 创建对象
p = create_point(10, 20);

# 调用 move() 方法
p.move(5, 3);

# 调用 print() 方法
p.print();
```

由于 crowbar 中没有专门的“方法”功能，因此通过上面的方式让对象的成员持有闭包，也就实现了与 Diksam、Java、C++ 等语言类似的方法功能。



代码清单 9-1 的第 3 行出现的 this 也不是关键字，其实只是取什么名字都可以的局部变量，只不过使用 this 的话，对于习惯了 C++ 或 Java 的人来说比较容易理解。重点在于，从闭包内部可以访问到外部的局部变量，因此在 print() 和 move() 的内部可以引用到 this。

如果想要继承或者多态的话，只需要在调用 create_point() 的基础上增加新的方法覆盖原有的方法就可以了。

代码清单 9-2
Point.crb（子类）

```
# 生成“子类”
function create_extended_point(x, y) {
    this = create_point(x, y);

    # 重写 print()
    this.print = closure() {
        print("**override** (" + this.x + ", " + this.y + ")\n");
    };

    return this;
}
```

另外，现在的 create_point() 方法外面如果书写代码 p.x 的话是可以引用到 x 的。也就是说，p 的成员 x、y 的默认是 public 的。如果实在是不喜欢这种方式，也可以不在 this 中保存 x 和 y，而是使用代码清单 9-3 的方式增加访问器就可以了。

代码清单 9-3
Point.crb（封装版）

```
# 创建“点”的函数（构件方法）
function create_point(x, y) {
    this = new_object();

    this.print = closure() {
        print("(" + x + ", " + y + ")\n"); # ←即使不是 this.x 也可以引用
    };
    this.move = closure(x_vec, y_vec) {
        x = x + x_vec;
        y = y + y_vec;
    };
    # 增加访问器（省略了 get_y()）
    this.get_x = closure() {
        return x;
    };
    return this;
}
```

在 create_point() 内创建的闭包拥有可以引用参数（或者说局部变量）x 和 y 的特性。

上面这些就是基于 crowbar 的面向对象。

9.1.5 闭包的实现

看了前一节的代码，可能有人会有下面这样的疑问。

> 如果 this、x、y 都是局部变量的话，在函数 create_point() 退出的时候不是就该被释放了吗？就算是在闭包中可以引用到外部的局部变量，但如果被释放了的话不是就不能引用了吗？

真是一个不错的问题，但上面的担心并不会发生，这才是闭包有趣的地方。

在 C 语言中，进入函数的时候会在栈上创建局部变量的内存空间，函数退出的时候会被释放。此时，被创建 / 释放的单块内存被称为**帧**（frame）。

在 crowbar 中，到 book_ver.0.3 为止，本质上都是相同的（只不过帧不是在栈上被创建的而是在堆上）。然而在上述示例中的 print() 和 move() 方法，在 create_point() 结束后才被调用，而且在方法里面还能引用到 this。如果使用“函数退出时帧也会被释放”这个老规则，是不能满足这种变量访问方式的。

在现在的 crowbar 中，创建帧的时机和往常一样，但是释放的时机却不是“函数退出的时候”，而是“帧不再被引用的时候”。也就是说，帧的释放是通过 GC 进行的。

但是，请思考一下实际的实现方式。

首先，帧是某一次函数调用时保存局部变量的地方，对于局部变量（群）来说，由于存在着多个变量名和值的组合，因此使用 assoc 再适合不过了。也就是说，在调用函数的时候创建一个 assoc，并将局部变量保存在其中就可以了。

接下来就是闭包了。像之前所说的，闭包具有如下特性。

1. 闭包是一个值，可以像C的函数指针一样赋值给变量，并且在之后可以通过函数调用的方式投入使用。
2. 可以引用创建其位置的局部变量。

首先，先来介绍一下第一个特征。

由于闭包是一个值，必然可以保存在 CRB_Value 中，因此，在 CRB_Value 的联合体定义中需要增加 CRB_Closure（代码清单 9-4）

代码清单 9-4
CRB_Closure 结构体

```
typedef enum {
    CRB_BOOLEAN_VALUE = 1,
    CRB_INT_VALUE,
    CRB_DOUBLE_VALUE,
    CRB_STRING_VALUE,
    CRB_NATIVE_POINTER_VALUE,
    CRB_NULL_VALUE,
    CRB_ARRAY_VALUE,
    CRB_ASSOC_VALUE,
    CRB_CLOSURE_VALUE,          ←新增
    CRB_FAKE_METHOD_VALUE,      ←将在后面的章节中介绍
    CRB_SCOPE_CHAIN_VALUE
} CRB_ValueType;
.
.
.
typedef struct {
    CRB_ValueType type;
    union {
        CRB_Boolean     boolean_value;
        int             int_value;
        double          double_value;
        CRB_Object      *object;
        CRB_Closure     closure;        ←新增
        CRB_FakeMethod  fake_method;    ←将在后面的章节中介绍
    } u;
} CRB_Value;
```

这里增加了 FAKE_METHOD，我会在后面的章节中介绍（抱歉，要在后面介绍的内容太多了）。

然后是第二个特性。闭包虽然与函数指针相似，但是它拥有可以引用创建其位置的局部变量的特性（对于这点，也有闭包保存了其创建位置的环境[①] 的说法）。局部变量被保存在 assoc 中，因此，我想 CRB_Closure 结构体可以定义成下面这样。

```
typedf struct {
    CRB_FuntionDefinition   *function;
    CRB_Object              *environment; /* 指向帧的 assoc */
} CRB_Closure;
```

成员 function 指向了函数定义的实体 CRB_FunctionDefinition，environment 则指向了创建闭包的位置的帧。

① 环境：environment。

但是，这里有一个必须要注意的地方，就是闭包是可以嵌套的。

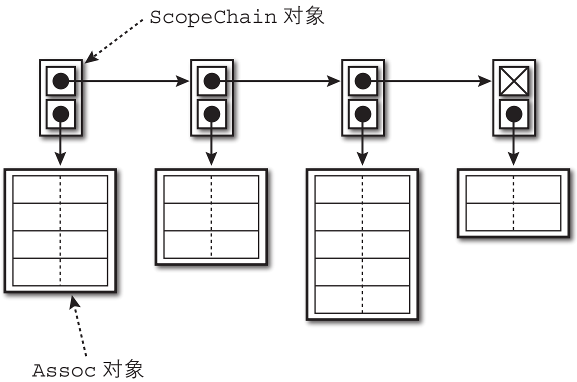
下面这段代码中，在注释的位置肯定可以同时引用到 a 和 b。

```
function f() {
    a = 10;
    c1 = closure() {
        b = 20;
        c2 = closure() {
            # 这里可以同时引用到 a 和 b
            print("a.." + a + "\n");
            print("b.." + b + "\n");
        };
        c2();
    };
    return c1;
}
```

虽说闭包是一个函数，但又不同于函数，因此它既可以访问保存了 a 的帧（函数 f() 的帧），又可以访问保存 b 了的帧（闭包 c1 的帧）。因此可以看出，闭包要保存的帧不止一个。

于是，引入了**作用链**（scope chain）这个概念。作用链是以链表方式管理的帧的 assoc（图 9-2）。

图 9-2
作用链



为了构建这个链表，我们引入了 ScopeChain 结构体。ScopeChain 作为 GC 的目标，也因此成为了 CRB_Object 的联合体的成员（这个是前面写到的要在后面章节中做介绍的内容之一）。

ScopeChain 结构体的定义如下。

```
typedef struct {
    CRB_Object   *frame; /* 指向 CRB_Assoc */
```

```
    CRB_Object   *next;  /* 指向 ScopeChain */
} ScopeChain;
```

但是，CRB_Closure 并没有直接指向代表帧的 assoc，而是指向了 ScopeChain。

```
typedef struct {
    CRB_FunctionDefinition *function;
    CRB_Object           *environment; /* 指向 ScopeChain */
} CRB_Closure;
```

不好意思各位，结果最后都是 CRB_Object，除了被注释的内容之外没有任何改变 *。

* 说到这里，各位一定会羡慕能够使用继承的语言吧。

另外，LocalEnvironment 结构体中也同样没有指向 assoc，而是指向了 ScopeChain。

至于具体怎么去使用上面定义的这些结构体，我想在跟踪程序实际执行时考虑的话，可能会更容易理解。

9.1.6 试着跟踪程序实际执行时的轨迹

在 crowbar 中，调用函数、创建闭包、调用闭包，都要进行以下动作。

[规则 1] 在调用函数的时候，会创建新的 LocalEnvironment 并将其入栈为栈顶 *。在这个 LocalEnvironment 中，为了保存在当前函数中声明的局部变量，创建并分配（元素被作为一个作用链）了一个新的 assoc。

[规则 2] 在创建闭包的时候，这个闭包保存着栈顶 LocalEnvironment 中的作用链。

[规则 3] 在调用闭包的时候，会在规则 1 中创建新的作用链之后，再将其链接到保存着闭包的作用链上。

* 在crowbar 中，LocalEnvironment 被创建在堆中，但实际上链表是在栈中实现的，因此在这里认为是创建在栈上的。

这些规则在实际上是怎样操作的，请参考代码清单 9-5。

代码清单 9-5
试着追踪闭包执行的轨迹

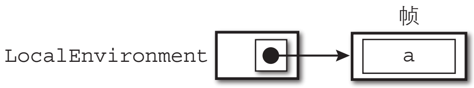
```
function f() {
    a = 10;
    c1 = closure() {
        b = 20;
        c2 = closure() {
            # 这里可以同时引用到 a 和 b
            print("a.." + a + "\n");
            print("b.." + b + "\n");
        };
        c2();
```

```
11:     };
12:     return c1;
13: }
14:
15: c = f();
16: c();
```

1. 普通的函数调用

首先，在第 15 行调用 f() 的时候，会根据规则 1 创建一个 LocalEnvironment，并生成第一个帧。在第 2 行的赋值语句中，声明了 a 并保存在新创建的帧中。之后的动作就跟调用一般的函数一样了（图 9-3）。另外，为了更清楚地描述这个过程，图中没有出现 ScopeChain 结构体，而是画得好像代表帧的 assoc 可以单独构建链表一样。

图 9-3
普通的函数调用

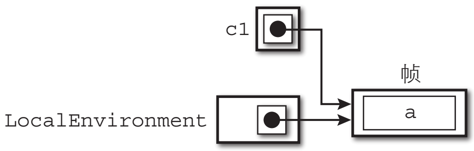


2. 创建闭包

第 3~11 行创建了闭包。

根据规则 2，闭包在创建的时候保存了指向保存了 LocalEnvironment 的作用链的引用。

图 9-4
创建闭包



并且，c1 本身就是一个局部变量，因此与 a 保存在了同一个帧中，图中为了使表达更为简单所以把这部分省略了。

这里的“创建闭包”是指，执行用关键字 closure 创建闭包的处理的时候。第 3 行中创建了闭包并赋值给了变量 c1，但是并没有马上调用闭包 c1，在第 5 行生成另外一个闭包的时候也没有执行，直到调用了 f() 将其返回，闭包 c1 一直都没有被调用。

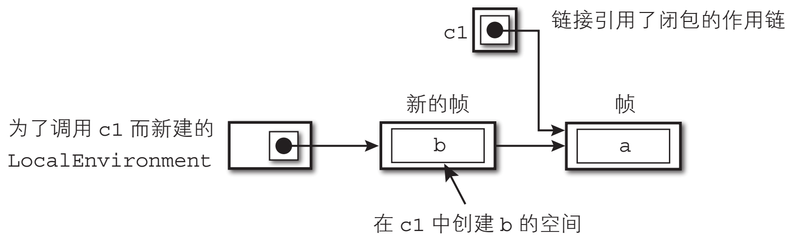
3. 调用闭包

调用 f() 返回之后，在第 16 行调用了闭包 c1。

此时也会根据规则 1 新建一个 LocalEnvironment 和帧，并且根据规则 3，

在新建了帧之后，再将其链接上保存着闭包的作用链（图 9-5）。

图 9-5
调用闭包



在搜索局部变量的时候，会对 LocalEnvironment 引用的作用链依次进行搜索。因此，在 c1 中是可以引用到局部变量 a 的。

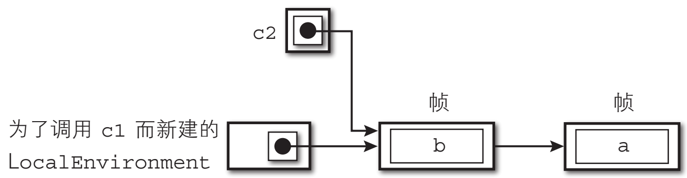
4. 创建闭包中的闭包

在开始执行赋值给 c1 的闭包时，首先会执行第 4 行为 b 的赋值。b 是一个单纯的局部变量，因此会在 LocalEnvironment 直接指向的帧中创建空间。

在接下来的第 5~9 行中创建第二个闭包 c2。

此时，根据规则 2，创建闭包 c2 时保存了指向 LocalEnvironment 保存着的作用链。因此，c2 保存的作用链中链接着存有 b 的帧和存有 a 的帧（图 9-6）。

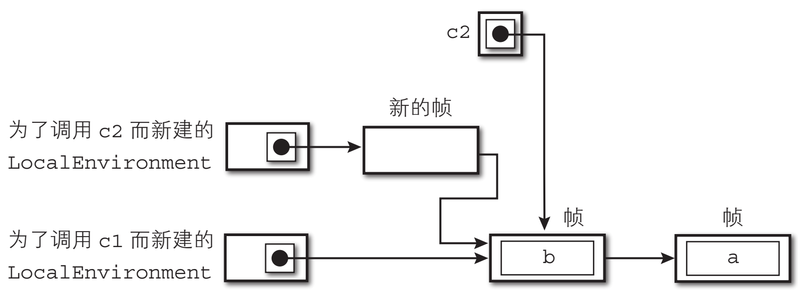
图 9-6
调用闭包



5. 调用嵌套闭包

在第 10 行调用了 c2 的时候，根据规则 3，c2 引用的作用链链接到了新的 LocalEnvironment 上，因此，c2 中应该可以同时引用 a 和 b。

图 9-7
调用嵌套闭包



9.1.7 闭包的语法规则

创建闭包的语言规则如下所示：

```
closure_definition
        : CLOSURE IDENTIFIER LP parameter_list RP block
        | CLOSURE IDENTIFIER LP RP block
        | CLOSURE LP parameter_list RP block
        | CLOSURE LP RP block
        ;
```

在 closure_definition 被 reduce 的时候，在 create.c 中会创建如下结构体（想象一下就能知道，它是 Expression 结构体的联合体成员）。

```
typedef struct {
    CRB_FunctionDefinition *function_definition;
} ClosureExpression;
```

由于闭包也是函数，因此为闭包表达式构建了 CRB_FunctionDefinition。但是这个 CRB_FunctionDefinition 仅可以由分析树中的 ClosureExpression 引用，并不会添加到 CRB_Interpreter 的 function_list 里面。

另外，在上面的四个语法规则中，有两个在关键字 closure 后面加入了标识符（IDENTIFIER）。这种语法规则在创建**命名闭包**时使用。

在目前为止出现的例子中，闭包都是没有名字的。但如果想要在闭包中递归调用自己的话，给闭包起个名字就会方便很多。

至于实现上，在调用命名闭包时，首先要为其创建新的帧，此时将闭包自身作为指定名称的局部变量登录进来就可以了。

9.1.8 普通函数

到 book_ver.0.3 为止，crowbar 的函数调用语法规则如下所示。

```
primary_expression
        : IDENTIFIER LP argument_list RP
        | IDENTIFIER LP RP
        ;
```

为了可以使用闭包，函数调用运算符 () 的左边不仅可以是 IDENTIFIER，还可以是任意的表达式。

因此，普通函数也发生了变化，函数名用来返回“表示函数的值”（C 语言中的函数指针）。函数的实体也变成了 ClosureExpression。

例如，调用函数 print("hello\n") 时的形式为，返回 print 标识符对应的“函数”，然后再调用它。此时，print 返回的 ClosureExpression 中，environment 成员为 NULL。

当然也可以通过下面的方法将普通函数赋值给变量。

```
p = print;
p("hello,world\n");
```

9.1.9 模拟方法（修改版）

在 crowbar 的数组中有例如 size() 这样的“方法”。

book_ver.0.2 的实现方式在 4.4.2 节中已经介绍过了，但是在这次的修改中，函数将作为“值”被处理。因此，像下面这样一段代码必须能够输出 array 的大小。

```
a = array.size;
⋮
print("array.size.." + a());
```

想要实现它，就必须要让 a.size 返回 C 语言中的函数指针——不仅如此，如果没有地方保存指向 array 的引用，也不可能返回数组的大小。

因此，在 CRB_Value 联合体中，增加了专门用来“模拟方法”的 CRB_FakeMethod 成员（这也是前面说过的会在后面章节中介绍的内容之一）。

```
typedef struct {
    char        *method_name;   /* 方法名 */
    CRB_Object  *object;        /* 相当于 this 的对象 */
} CRB_FakeMethod;
```

CRB_FakeMethod 结构体保存了前面所说的指向 array 的引用。在知道了这些引用和方法名之后，处理器就可以调用模拟方法了。

9.1.10 基于原型的面向对象

实际上，crowbar 的设计方式多多少少参考了 JavaScript。

像 JavaScript 和 crowbar 这样没有类的概念、每个实例都包含不同字段和方法的语言，被称为**基于原型的面向对象语言**（prototype based object oriented language）。与此相对，Java、C# 和 Diksam 等具有类的概念的面向对象被称为**基于类的面向对象语言**（class based object oriented language）。

但是，一般被称为“基于原型的面向对象”时，多数情况会包含如下特性：

- 可以通过复制（克隆）现有对象来创建新的对象。
- 当调用了对象的某个方法之后，如果对象中不存在这个方法，则自动将该方法的调用传递给其他对象[1]（**原型链**，prototype chain）。

虽然在 JavaScript 中具有原型链功能，但是在 crowbar 中却没有。这也就意味着，crowbar 算不上是基于原型的面向对象语言。但是，基于原型的面向对象也被称为是“基于实例的”或者是“基于对象的”，从这两个方面去看 crowbar，也许还能够跟它们归为一类。

9.2 异常处理机制

在现在的语言中，当程序运行过程中发生了预期之外的情况时，一般会发生**异常**（exception）。在 C 语言中，多数情况下会通过返回值不停地给调用者返回“错误状态”。这种做法不仅很麻烦，还会使代码由于嵌入了异常处理程序而变得难以阅读。

因此，我考虑在 crowbar 和 Diksam 中引入异常的概念。

9.2.1 为 crowbar 引入异常

在 crowbar 中引入了与 Java 相同的 `try~catch~finally` 处理方式。

---

① 当然这两个对象要具有原型继承关系。——译者注

在 Java 和 C# 的 catch 子句中虽然可以对应不同的异常类型并进行处理，但是因为 crowbar 中本来就没有类型，所以也就只写一个 catch 子句。具体的示例请参考代码清单 9-6。

代码清单 9-6
crowbar 的异常处理示例

```
 1: try {
 2:     zero = 0;
 3:     a = 3 / zero;
 4: } catch (e) {
 5:     # 通过 child_of 方法判断异常种类
 6:     if(e.child_of(DivisionByZeroException)) {
 7:        print(" 不能被 0 除。\n");
 8:     } else {
 9:        throw e;
10:     }
11: }
```

代码清单 9-6 的第 3 行试图用 3 除以 0，这样做会发生被 0 除的异常（这里特意使用变量 zero 的原因是，如果直接使用 3/0 的话在编译时会出现错误）。

第 4 行的 catch 子句捕捉了这个异常。

第 6 行中，通过调用捕获异常对象的 child_of() 方法来检查异常的类型。这段代码里检查了异常是否是 DivisionByZeroException 类型，如果是则输出错误信息。

如果不是处理器定义的异常，而是自己认为发生了异常情况，可以使用 throw 抛出自定义异常。

```
e = new_exception(" 错误信息 ");
throw e;
```

new_exception() 是一个原生函数。返回值为异常对象，是一个 assoc。调用 new_exception() 时，会在返回值中保存**栈轨迹**（stacktrace）。这个异常如果没有被 catch 的话，会一直传播到顶层结构。处理器会记录栈轨迹，也可以通过 print_stack_trace() 方法从程序中输出栈轨迹。

像被 0 除这种在处理器中发生的异常通常都会有父子关系，例如 DivisionByZeroException 就是 ArithmeticException 的子类。因此，在代码清单 9-6 的第 6 行可以将 DivisionByZeroException 替换为 ArithmeticException，child_of() 方法仍然会返回真。

通过在各“异常类”中保存父异常的方式来实现这种父子关系。另外，当然无论是 ArithmeticException 还是 DivisionByZeroException 都不是

关键字，只是全局变量而已。说了这么多，还是快点来看一段代码吧（代码清单 9-7）。

代码清单 9-7
“异常类”的实现

```
 1: function create_exception_class(parent) {
 2:     this = new_object();
 3:     this.parent = parent;
 4:     this.create = closure(message) {
 5:         e = new_exception(message);
 6:         e.stack_trace.remove(0);
 7:         e.child_of = this.child_of;
 8:         return e;
 9:     };
10:     this.child_of = closure(o) {
11:         for (p = this; p != null; p = p.parent) {
12:             if (p == o) {
13:                 return true;
14:             }
15:         }
16:         return false;
17:     };
18:     return this;
19: }
20:
21: RootException = create_exception_class(null);
22: BugException = create_exception_class(RootException);
23: RuntimeException = create_exception_class(RootException);
24: ArithmeticException = create_exception_class(RuntimeException);
25: VariableNotFoundException = create_exception_class(BugException);
    （之后省略）
```

代码清单 9-7 的源文件在 buitin 目录下的 builtin.crb 中。具体是如何加载它的将在 9.3 节中介绍。

从第 1 行开始的 `create_exception_class()` 函数是“异常类”的构造函数。各种异常类型从第 21 行开始被定义为全局变量。异常类对于程序来说只存在一个，可以通过调用异常类的 `create()` 方法来创建这个异常类的实例。

父异常通过“异常类”的构造函数接收的参数 `parent` 被保存起来。因此，第 10 行的 `child_of()` 方法可以检查异常的层级。只是，由于 `child_of()` 是“异常类”的方法，因此在创建异常的实例时要把它设置到异常的实例中（第 7 行）。异常实例中的 `parent` 成员即使什么都没有，也可以调用 `child_of()` 方法，这就是闭包的魔力。

另外，第 6 行移除了栈轨迹的第一个元素，这样做是为了不让栈轨迹中包含

调用 `create()` 方法的痕迹。

在 crowbar 中，异常的栈轨迹以下面的格式输出。

```
不能被 0 除。
func at 6
func at 3
func at 3
func at 3
top_level at 9
```

上面这段输出来自代码清单 9-8。在递归调用 `func()` 时，`DivisionByZeroException` 异常会发生在程序的深处。

代码清单 9-8
exception.crb

```
function func(count) {
    if (count < 3) {
        func(count + 1);
    }
    zero = 0;
    a = 3 / zero;
}

func(0);
```

9.2.2 setjmp()/longjmp()

crowbar 的程序是一边递归分析树一边执行的。因此，在发生异常的时候，会一下子追溯到 C 语言的调用层级中。

crowbar 的控制结构 `return`、`break`、`continue` 有着同样的问题，在使用上述控制结构时，会通过返回值将各种状态返回给调用者。但是，相对于 `return` 和 `break` 只会发生在“语句”级别，异常有可能发生在“表达式”的深处，因此要将它对应返回值会比较麻烦。这种情况下，我们可以使用 `setjmp()/longjmp()`。

普通的 C 语言使用者可能大多数还不太熟悉 `setjmp()/longjmp()`。更确切地说，有的人认为“它是一个比 goto 还要邪恶的，可以跨越函数界限进行长距离（long）跳转（jmp）的可怕函数！”

但是，无论什么事情，无论是好是坏，都要好好的研究一下才能下结论。如果自己连用都没用过就说“这个不好用”，那这人也实在是荒唐。

所以，以下我们先简单地看一下 `setjmp()/longjmp()`。

- setjmp() 的参数是 jmp_buf 类型的变量，调用函数时在参数中保存程序的上下文。
- longjmp() 用于返回到使用 setjmp() 保存的位置。

即使通过多次函数调用进行到了很深的级别，也可以瞬间返回。

从 longjmp() 这个名字就可以看出来，它可以跨越函数界限随意跳转到任何位置。但实际上，它只能“返回”到使用 setjmp() 标记过的位置。在 longjmp() 看来，setjmp() 标记的必须是“调用者”。在 longjmp() 的时候，被 setjmp() 标记了的函数在没有返回的情况下会从栈中删除。基于以上操作，我觉得这个方法叫 longjmp() 有点不太合适，应该叫 longreturn() 才更加贴切。

更重要的是，setjmp() 在保存程序上下文的时候返回 0。在使用 longjmp() 返回时，longjmp() 的第 2 个参数也会跟着返回。根据第 2 个参数返回的值，可以判断出当前执行的程序是从哪个 longjmp() 返回的。

使用这两个函数编写下面这段代码的时候，可以从深层的函数调用中瞬间返回回来。

```
/* 为了保存调用者的程序上下文，声明了一个变量 */
jmp_buf recovery_environment;

if (setjmp(recovery_environment) == 0) {
    /* setjmp() 在第一次调用的时候返回 0，进入这个分支
     * 在这个分支中进行正常情况下的处理
     * 在这里调用了 longjmp() 后，会执行下面的 else 子句。
     * 对于 longjmp() 的调用，在这里即使进行了
     * 深层次的函数调用，其结果也不会改变。
     */
    long_jmp(recovery_environment, 1);
} else {
    /* 执行了 longjmp() 之后会进入这个分支进行处理
     */
}
```

作为参数被传入 setjmp() 的 jmp_buf 类型变量中保存着“当前程序的上下文”。“当前程序的上下文”中包含了当前寄存器的值等很多东西，但首当其冲要记录的我认为就是 setjmp() 被调用的地点。再把这个 jmp_buf 当做参数传递给 longjmp() 的话，就应该能返回到 jmp_buf 记录的地点了。

另外，可能有人会有这样的疑问：“setjmp() 的参数并没有加上 & 传递，为什么还能保存程序的上下文呢？ C 的参数不是按值传递的吗？”这是因为 jmp_

buf 类型被 typedef 成了数组。请务必检查您环境的头文件。我觉得这是一个会招致混乱的设计。

实际上 crowbar 的异常处理在代码清单 9-9 中进行了实现（execute.c）。

代码清单 9-9 execute_try_statement() 函数

```
static StatementResult
execute_try_statement(CRB_Intercepter *inter, CRB_LocalEnvironment *env,
                      Statement *statement)
{
    StatementResult result;
    int stack_pointer_backup;
    RecoveryEnvironment env_backup;

    /* 备份 crowbar 栈的栈指针和 jmp_buf */
    stack_pointer_backup = crb_get_stack_pointer(inter);
    env_backup = inter->current_recovery_environment;
    if (setjmp(inter->current_recovery_environment.environment) == 0) {
        /* 执行 try 子句 */
        result = crb_execute_statement_list(inter, env,
                                            statement->u.try_s.try_block
                                            ->statement_list);
    } else {
        /* 发生异常时的处理。首先恢复 crowbar 的栈和 jmp_buf */
        crb_set_stack_pointer(inter, stack_pointer_backup);
        inter->current_recovery_environment = env_backup;

        if (statement->u.try_s.catch_block) {
            /* 执行 catch 子句 */
            CRB_Value ex_value;

            ex_value = inter->current_exception;
            CRB_push_value(inter, &ex_value);
            inter->current_exception. type = CRB_NULL_VALUE

            assign_to_variable(inter, env, statement->line_number,
                               statement->u.try_s.exception, &ex_value);

            result = crb_execute_statement_list(inter, env,
                                                statement->u.try_s.catch_block
                                                ->statement_list);
            CRB_shrink_stack(inter, 1);
        }
    }
    inter->current_recovery_environment = env_backup;
    if (statement->u.try_s.finally_block) {
        /* 执行 finally 子句 */
        crb_execute_statement_list(inter, env,
```

```
43:                                        statement->u.try_s.finally_block
44:                                        ->statement_list);
45:     }
46:     if (!statement->u.try_s.catch_block
47:         && inter->current_exception.type != CRB_NULL_VALUE) {
48:         /* 如果没有 catch 子句的话，创建新的 throw 直接抛出异常 */
49:         longjmp(env_backup.environment, LONGJMP_ARG);
50:     }
51:
52:     return result;
53: }
```

crowbar 使用自己的栈计算表达式，在第 10 行备份了这个栈。因为如果表达式执行到深处时发生了异常的话，就必须要抛弃那个时候的栈。

另外，在第 11 行使用变量 evn_backup 进行了备份，这个变量的类型是 RecoveryEnvironment，因此它的实体只包含了一个 jmp_buf 的结构体（这里特意声明了一个结构体以备将来扩展）。

这两个用于备份的局部变量被放在了 C 的栈上。因此，没有 catch 子句或从 catch 子句中再 throw 后，无论返回到任何阶段（第 49 行），都需要依次从 C 栈上的备份中恢复这两个变量。

并且，无论从 try 子句中调用什么阶段的函数，如果在深层发生异常，是不能瞬间返回到 catch 中，而是顺着函数调用的顺序返回的。因此，在调用 crowbar 函数的时候也会执行 setjmp()。这样做的前提是 CRB_LocalEnvironment 在必要的时候能够开放（基于 eval.c 的 eval_function_call_expression()）。

```
stack_pointer_backup = crb_get_stack_pointer(inter);
env_backup = inter->current_recovery_environment;
if (setjmp(inter->current_recovery_environment.environment) == 0) {
    do_function_call(inter, local_env, env, expr, func);
} else {
    /* 如果在函数内发生异常的话，抛弃 LocalEnvironment */
    dispose_local_environment(inter);
    crb_set_stack_pointer(inter, stack_pointer_backup);
    /* 紧接着调用 longjmp() */
    longjmp(env_backup.environment, LONGJMP_ARG);
}
inter->current_recovery_environment = env_backup;
dispose_local_environment(inter);
```

另外，在代码清单 9-9 的第 47 行检查了 inter->current_exception.type，这样做是为了保证它保存的是“当前的异常”，以便在 throw 的时候进行

设置。因此，它会在 catch 子句中被抛弃（第 28 行），并在 catch 子句执行（第 33~35 行）并且发生异常的时候被重新设置。

再来说说 finally 子句，一旦进入了 try 子句就“必须”要执行 finally 子句。例如在下面这段代码中使用 break 跳出了 for 语句，即使是在这种情况下，在跳出循环之前也必须要执行 finally 子句。

```
for (;;) {
    try {
        break;
    } finally {
        # 即使进行了 break，这里的代码也会执行
    }
}
```

在 crowbar 中出现了 break 等跳转语句时，将把语句的执行结果（CRB_StatementResult）作为返回值返回给调用者（请参考 3.3.6 节）。在代码清单 9-9 中，无论 try 子句得到怎样的执行结果都会执行 finally 子句，以保证“必须执行 finally”。但是，另一方面，如果 try 子句中进行了 break 的话，在 finally 执行结束后还是要执行 break。还有，如果在 try 中 return 3;、在 finally 中有 return 5; 的时候，到底要返回哪个才对呢？在 Java 中，如果在 finally 子句中写了 return、break 等控制语句的话，javac 会发出警告。C# 中则会直接发生编译错误。

在 crowbar 中，try 语句最终执行结果的原则是，无论 finally 中是什么结果都会被忽略，要优先使用 try 或者 catch 子句的执行结果。

补充知识　**Java 和 C# 异常处理的不同**

关于 Java 和 C# 异常处理的不同点，通过搜索引擎搜索“Java C# 异常 不同”这样的关键字就能得到“有无检查异常”的相关资料。

这点会在 9.2.6 节的补充知识中做介绍。除此之外，C# 的异常和 Java 的异常还有很多区别，Java 的程序员如果使用 C# 的话会很容易上手（反过来就不一定了）。

首先，在 Java 中，异常的栈轨迹创建于“异常 new 出来的时候”。比如下面这段程序中，异常一被 new 出来后马上就输出栈轨迹，此时可以把当前的栈轨迹完全输出出来。

```
Exception e = new Exception();
e.printStackTrace();
```

多数人可能会在调试的时候编写这样的代码。与此相对在 C# 中，栈轨迹在

“throw 之后返回到方法调用的层级的时候”被依次组装起来的。可以通过代码清单 9-10 的示例代码来确认这个问题。

代码清单 9-10
C# 的异常

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ExceptionTest
{
    class Program
    {
        static void Sub(int count)
        {
            if (count == 0)
            {
                throw new Exception();
            }
            try
            {
                Sub(count - 1);
            }
            catch (Exception e)
            {
                // 进行递归调用使异常发生在较深的层级
                // 每一个层级 catch 到异常后，再继续 throw。
                // 在每次 catch 的时候，栈轨迹都会随之增长，从这点就可以看出
                // C# 中异常的栈轨迹，是在返回到调用者的层级后
                // 再进行组装的。
                Console.WriteLine("********** count.." + count + "*********");
                Console.Write(e.ToString());
                throw;
            }
        }

        static void main (string[] args)
        {
            try
            {
                Sub(10);
            }
            catch (Exception e)
            {
                Console.WriteLine("********** final *********");
                Console.Write(e.ToString());
```

```
42:                }
43:            }
44:        }
45: }
```

说到这样做的理由，比如创建异常在某个工具类中的时候，如果栈轨迹中只包含了这个工具类的方法的话，就会让调试人员摸不着头脑。实际上，crowbar 也采用 Java 的方式（见代码清单 9-7），为了从栈轨迹中删除第 4 行的 create() 方法，不得不在第 6 行做那样奇怪的事情。

在 C# 中，调试时如果想看一下栈轨迹的话，可以使用 Environment.StackTrace。

另外，在 Java 中被 catch 到的异常如果想再次抛出的话，要编写下面这段代码：

```
} catch(HogeException e)  {
      ⋮
    throw e;
}
```

但是，在 C# 中 throw 的时候，会重新设置 e 中的栈轨迹，从而在 C# 中只需要这样写就可以了：

```
throw;
```

C# 中增加了“将在 catch 子句中捕获的异常直接抛出”的语法。

在 crowbar 中既然采用了 Java 的方式，那么在 Diksam 中让我们来试试看 C# 的方式。具体的实现方式将在下一节中介绍。

9.2.3 为 Diksam 引入异常

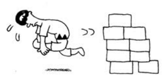
说完了 crowbar，本节就要为 Diksam 引入异常的概念了。

下面就为 Diksam 引入与 Java 和 C# 相同的异常处理机制。

```
try {
    /* try 子句 */
} catch (HogeException e) {
    /* 与 HogeException 对应的 catch 子句 */
} catch (PiyoException e) {
    /* 与 PiyoException 对应的 catch 子句 */
} finally {
    /* finally 子句，这里的代码必然会执行 */
}
```

由于 crowbar 中没有类的概念 *，因此只能编写一个 catch 子句。但是在

* 实际上是可以做出相似的异常层级结构的，即使使用 crowbar 的功能做出了这样一个层级结构，从语言的设计层次来讲并没有特别支持这种方式，而对我本人来说也很抗拒这种上下相逆的事情。



Diksam 中与 Java 相同，可以编写与各种异常类对应的 catch 子句。

另外，和 Java、C# 等相同的地方是，也可以通过 throw 抛出异常。

```
throw e; // throw 异常 e
```

crowbar 使用了 Java 的风格，即在 new 异常的时候创建栈轨迹，因此在 Diksam 中要使用 C# 的风格，即在 throw 的时候创建栈轨迹（请参考 9.2.2 节的补充知识）。所以，在 catch 子句中抛出异常的时候可以只写 throw;。

另外，和 Java、C# 等相同的可以被 throw 和 catch 的只有 Exception 的子类 *。

* 由于 Exception 是在 Diksam 中创建的类，这里果然还是做了“上下相逆”的事情。

补充知识　catch 的编写方法

如前面所述，在 crowbar 中只能编写一个 catch 子句，但是在 Diksam 中就可以编写与各种异常类对应的 catch 子句。“可以编写”看上去是一个很方便的功能，但是实际使用起来就不是那么方便了。

例如，异常 A 和异常 B 想要进行同样的处理，但是这种时候根据 Java 的风格，就只能把同样的 catch 子句编写多次了。这样一来就违反了“同样的代码不能在多个位置编写”的编码大原则。

当然，如果异常 A 和异常 B 用同样的超类的话，可以通过 catch 超类的方式达到目的，但是异常的层级通常都是从提供异常类的角度、而不是从异常使用者的角度出发的。

假设“想要对异常 A 和异常 B 进行同样的处理”的时候，就必须要简单地描述 OR 条件，只有这样才可以考虑在 catch 子句中用逗号分隔的方式指定要处理的类（先忽略如何将异常赋值给变量的事情）。还有一种情况，那就是“异常 A 和异常 B 使用同样的处理方式，异常 C 使用另外的处理方式，但无论是什么样的异常，都会有输出日志的通用处理”。基于上面这些考虑，我想还是像下面这样，在一个 catch 子句中使用 if 语句来判断可能更好。

```
try{
      ⋮
} catch (Exception e) {
    // 通用处理
    if (e instanceof A || e instanceof B) {
        // 处理异常A或者是异常B
    } elsif (e instanceof C) {
        // 处理异常C
    } else {
        // 预想外的异常向上throw
        throw;
    }
}
```

但这个方法要想处理发生的意料之外的异常，就必须要在 if 语句的 else 子句中继续 throw 异常。这段代码非常容易被忘记。

更重要的是，使用者既可以像 Java 那样编写多个 catch 子句，也可以像上面的代码那样编写一个 catch 子句，但如果只写一个 catch 子句的话，编码方式就不能像 Java 一样了。这意味着给了使用者更多的选择，在这点上 Diksam 和 Java 是一致的。

9.2.4 异常的数据结构

try 语句中包含了 try、catch 和 finally 三个子句。用 DVM_Try 结构体表示的话如下所示。

```
/* catch 子句 */
typedef struct {
    int class_index; /* 使用了 catch 的类的索引值 */
    int start_pc;    /* catch 开始位置的 PC（程序计数器）*/
    int end_pc;      /* catch 结束位置的 PC */
} DVM_CatchClause;

/* try 语句 */
typedef struct {
    int             try_start_pc; /* try 开始位置的 PC */
    int             try_end_pc; /* try 结束位置的 PC */
    int             catch_count; /* catch 子句的数量 */
    DVM_CatchClause *catch_clause; /* catch 子句的可变长数组 */
    int             finally_start_pc; /* finally 开始位置的 PC */
    int             finally_end_pc; /* finally 结束位置的 PC */
} DVM_Try;
```

在 DVM_Try 结构体的可变长数组中可以添加顶层结构和 DVM_Function。另外，由于顶层结构和函数（的程序块）增加了一些附加信息，因此这次把“程序块”也作为结构体抽取了出来。

```
typedef struct {
    int             code_size;
    DVM_Byte        *code;
    int             line_number_size;
    DVM_LineNumber  *line_number;
    int             try_size;   /* try 子句的数量 */
    DVM_Try         *try;       /* try 子句的可变长数组 */
    int             need_stack_size;
} DVM_CodeBlock;
```

虽然除了新增的 try 子句的相关成员外，其他成员都在以前的 DVM_

Executable（顶层结构部分）和 DVM_Function（各函数部分）中出现过了，但这里还是应该把它作为单独的结构体区分出来。DVM_Try 中的数组（在 generate.c 中）在后续遍历（post-order traversal）分析树时，每当遇到 try 语句的时候就在数组末尾增加元素（generate_try_statement()）。因此，这个数组是按照程序层级的深度来排列 try 语句的（越深层级的 try 就越被排在前面）。

在发生异常的时候，首先将这个异常设置到 DVM_VirtualMachine 结构体的 current_exception 成员（新增）中，再将 DVM 设置为“异常状态”。

基于以上介绍，我们以从 0 开始按顺序扫描 DVM_Try 的数组，寻找包含这个位置的程序计数器的 try 语句。结果有以下几种情况。

1. 异常发生在try语句的try子句中
   在这种情况下，首先要寻找捕捉已发生异常的catch子句，如果发现了，就解除异常状态并将控制权移交给相应的catch子句。
   如果没有与异常对应的catch子句，就把控制权移交给finally子句。
2. 异常发生在try语句的catch子句中
   在这种情况中，不解除异常状态并将控制权移交给finally子句。
3. 异常发生在try语句的finally子句中
   在这种情况中，和4.的处理方式相同。
4. 异常发生在try语句之外
   强制从当前函数中返回，并试图从基于当前位置的DVM_Try中寻找包含当前程序计数器的try语句，以此方式修正异常。

Diksam 的编译为了处理起来简单，不论 try 语句中是否有 finally 子句，都将为它创建一个 finally。这个将在后面介绍。

这个处理的代码如代码清单 9-11 所示。第 14 行调用的函数 throw_in_try()，会在上述 1、2 的情况下返回真。

代码清单 9-11
发生异常时的处理

```
1: static DVM_Boolean
2: do_throw(DVM_VirtualMachine *dvm,
3:          Function **func_p, DVM_Byte **code_p, int *code_size_p,
           int *pc_p,
4:          int *base_p, ExecutableEntry **ee_p, DVM_Executable
           **exe_p,
5:          DVM_ObjectRef *exception)
6: {
7:     DVM_Boolean in_try;
8:
```

```
    dvm->current_exception = *exception;

    for (;;) {
        /* 当异常发生在 try 语句的 try 子句或 catch 子句的时候，
            throw_in_try 函数会将设置跳转地址并返回真。*/
        in_try = throw_in_try(dvm, *exe_p, *ee_p, *func_p, pc_p,
                              &dvm->stack.stack_pointer, *base_p);
        if (in_try)
            break;

        if (*func_p) {
            /* 当异常发生在 finally 子句或者 try 语句外的时候，
               把发生异常的位置记录到栈轨迹中并强制返回。
               在返回之后，将再次使用 throw_in_try 进行修复。*/
            add_stack_trace(dvm, *exe_p, *func_p, *pc_p);
            /* do_return 在要程序返回原生函数时返回真，这里不做详细说明。*/
            if(do_return(dvm, func_p, code_p, code_size_p, pc_p,
                         base_p, ee_p, exe_p)){
               return DVM_TRUE;
            }
        } else {
            /* 在返回到顶层结构的时候将栈轨迹输出并终止程序的执行。 */
            int func_index
                = dvm_search_function(dvm,
                                      DVM_DIKSAM_DEFAULT_T_PACKAGE,
                                      DIKSAM_PRINT_STACK_TRACE_FUNC);
            add_stack_trace(dvm, *exe_p, *func_p, *pc_p);

            invoke_diksam_function_from_native(dvm, dvm->function[func_index],
                                               dvm->current_exception, NULL);
            exit(1);
        }
    }
    return DVM_FALSE;
}
```

9.2.5 异常处理时生成的字节码

首先，在使用者编写了像 throw e; 这样的代码在抛出异常的时候，这里将会创建 throw 指令。编译器会将眼前的 e 入栈，因此这个指令会抛出保存在栈顶的异常。

Diksam 的异常与 C# 风格相似，只需要写 throw; 就可以将当前 catch 的异常抛出去（请参考 9.2.2 节的补充知识）。在这个时候会创建指令 rethrow，但

是在 rethrow 指令中依旧不变地抛出保存在栈顶的异常（编译器会悄悄地把在 catch 子句中定义的变量入栈）。和 throw 动作不同的只是不会重新设置栈轨迹。

try 子句通常会生成下面这样的字节码（为了方便说明，使用了仿真代码，并在左边添加了行号）。

```
 1: # 在这里加入 try 子句的指令
 2: go_finally 14
 3: jump 16
 4: # 第一个 catch 子句
 5: pop_stack_object n # 将异常赋值给变量
 6: # 这里插入 catch 子句的指令
 7: go_finally 14
 8: jump 16
 9: # 第二个 catch 子句
10: pop_stack_object n # 将异常赋值给变量
11: # 这里插入 catch 子句的指令
12: go_finally 14
13: jump 16
14: # 这里插入 finally 子句的指令
15: finally_end
16: # 之后的处理
```

首先，请看一下每个 catch 子句开头的 pop_stack_object。这是为了在下面这种情况时，把异常的引用赋值给变量 e。

```
} catch (HogeException e) {
    ⋮
}
```

也就是说，DVM 将异常入栈为栈顶并将控制权移交给各 catch 子句。当然，e 作为函数外的变量被 pop_static_object 创建。

在 try 子句、catch 子句的末尾都要生成 go_finally 指令。这意味着通过在一个函数内部调用子例程的方式来调用 finally*。

* 这个机制模仿了 JVM 的指令 jsr、return，但是在现在的 Java 中并没有使用，而是将 finally 子句的代码完全展开。在 Sun 的错误数据库（Bug Database）中的 4381996 号错误[①]使用原来的方式时，即使是正确的代码，验证器也会报错。

乍看之下，我认为不需要特意增加这个指令，只要跳转到第 14 行就可以了。但是，正如 9.2.2 节中写到的，即使在 try 子句中进行了 break 或者 return，finally 子句也必然会执行。因此，如果 try 子句中包含了 break，编译器也会在 break 前面输出 go_finally 指令。执行 finally 子句后，如果没有异常状态的话，就会执行 finally_end 指令返回到原来的位

① 地址：http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4381996。——译者注

置。如果调用了 finally_end 后不能返回到原来的位置的话，就会执行和再次抛出异常时同样的动作。

go_finally 会将返回的目的地的 pc 入栈。之所以必须要使用栈是因为在 finally 中也可以编写 try 语句。

9.2.6 受查异常

Java 具有**受查异常**的功能。这是一种在方法声明时对方法可能抛出的异常进行声明的功能（下面这段代码就表明了“这个方法有抛出 HogeException 和 PiyoException 的可能”）。

```
void hoge() throws HogeException, PiyoException {
    ⋮
}
```

这样一来，在调用上面的 hoge() 方法的时候，对于该方法的调用者来说，要么 catch 所有已经被声明的异常，要么自己也通过 throws 抛出这些异常，将处理异常的任务交给自己的调用者。如果你什么都不做，就会在编译时报错（除非是 Error 或者 RuntimeException 的子类）。Java 利用这个功能保证应该处理的异常已经全都被处理掉（至少是以此为目标）。

正如后面会介绍到的，对于这个功能也有一些异议，但我认为这是一个重要的功能，因此在 Diksam 中也进行了实现。

Diksam 中受查异常的设计和 Java 相同。

首先，要让函数和方法能够描述 throws 子句。

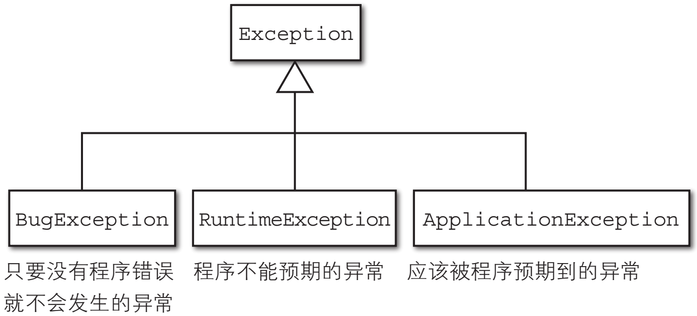
```
// 为函数添加 throws
void func() throws HogeException, PiyoException {
    ⋮
}
```

在上述例子中，由于 func() 已经声明了它可能会发生 HogeException 和 PiyoException，这时候如果在函数内发生了上述两个异常之外的异常，并且又没有 catch 的话，就会发生编译错误。

但是，像 NullPointerException 这样的在程序各处都会发生的异常应该被另行处理。在 Diksam 中异常的层级有三个，它们分别是 BugExce

ption、RuntimeException 和 ApplicationException。其中，只有 ApplicationException 是受查异常检查的对象（图 9-8）。

图 9-8
Diksam 的异常层级



例如，引用了 null 的时候会发生 NullPointerException，或者当访问的元素超出了数组的返回时会发生 ArrayIndexOutOfBoundsException，这些异常在 Diksam 中都被归类为 BugException。这是因为这些异常在调试结束后的正式应用程序中是不应该发生的（我是这么认为的）。因此，BugException 不应该被 catch。因为如果这么做就相当于在“掩盖程序错误”。

与此相对，即使没有程序错误也可能发生 RuntimeException，这是由于在写程序的时候只考虑了一般的情况，没有考虑周全，从而发生了（我是这么认为的）异常。在 Diksam 中，整数被 0 除也被归为这类异常 *。这样的异常应该在调用的层级上被 catch 并进行适当地处理。

* 我本来想把内存不足之类的情况也加入到异常中，但是在现在的 Diksam 实现中，只有在 MEM_malloc() 中进行 exit() 的时候才会抛出异常，MEM_malloc() 本身并不会发生异常。

ApplicationException 的发生是被充分预期的。在 Diksam 中，像 NumberFormatException 这样的异常被归入此类 *。像这样的异常一般会在发生的地方立即由应用程序做适当地处理。

* 不可思议的是，Java 把这个异常归类为 RuntimeException。

话说回来，由于在 Diksam 中 BugException “不应该被 catch”，因此，还是“一旦 catch 了 BugException 就会发生编译错误”容易一点。但是现在之所以没有这么做，是因为考虑到了像 Servlet 和 Applet 之类的在浏览器中运行的程序，即使其中一个 Servlet 或者 Applet 中出现了程序错误也不应该导致整个程序的崩溃。但是，在将来也可能会引入像 pragma 这样的功能，也许到了那个时候，除了特殊的程序之外，再 catch 了 BugException 的话，编译就要报错了。

受查异常的实现在 fix_tree.c 中进行。

Diksam 编译器（在 fix_tree.c 中）会对递归分析树“确认”（请参考 6.3.4 节）。与此同时，也会进行受查异常的检查。

在递归扫描分析树的时候，在递归的路径上（也就是越深越优先）会创建语句或表达式下级可能会发生的异常列表。此时，在 try 子句中发生的异常，除了会被 catch 子句捕捉之外，剩下的都会被当做是 try 语句发生的异常 *。这样一来，在函数内发生异常的时候，没有声明 throws 的就会导致编译错误。

* 详细来说就是，这里涵盖了在 catch 子句或 finally 子句中发生的异常（含通过 throw; 再次抛出的 Exception）。

补充知识　受查异常的是与非

在 Java 中有受查异常，但是在 C# 中没有。C# 是晚于 Java 面世的编程语言，因此很多地方都模仿了 Java，所以可以断定这个功能是有意被剔除的。关于这点，C# 的作者安德斯・海尔斯伯格（Anders Hejlsberg）例举了下面两个理由（应笔者邀请）。

The Trouble with Checked Exceptions[11]

- 方法升级的时候 throws 子句中的异常可能也会随之增加，这样就会影响所有的使用者。实际上，在很多情况下，调用者并不会关心异常的种类，也不会对它们做个别处理。
- 在扩展性上存在问题。受查异常在很小的程序中可以顺利运行，但是在构建有 4 个到 5 个子系统的系统时，每个子系统又返回 4~10 种异常的话，在多个子系统集成的时候就不得不在 throws 后面写上很多的异常。

另外，在下一页中，在上面这些理由的基础上又追加了以下理由（这里也是应笔者邀请）。

《Java 理论与实践：关于异常的争论 要检查，还是不要检查》[12]

- throws 子句暴露了实现的详细内容。如果在“搜索用户”方法的 throws 中有一个 SQLException，这可不是一件好事。
- （只要是有受查异常）使用者编写了空的 catch 子句，异常也会被当做处理掉了。

上面说了这么多的问题，其实可以通过异常包装的方式来应对。以“搜索用户方法”为例，应该抛出像 NoSuchUserException 这样的、对于当前方法层级有意义的异常。从而，如果发生异常的原因是 SQLException 的话，应该抛出一个以成员形式保存了 SQLException 的 NoSuchUserException。

这个机制，在 Java1.4 中被引入，C# 一开始也有这样的功能。以此为前提的话，我认为受查异常也不是特别“坏”的功能 *。

* 在 [12] 中也举出了诸如“异常过度包装”之类的缺点，也算是表达了对这个功能的不满。

另外，在受查异常中最让人反感的就是，所有方法都只写一句 throws Exception。对于这点，虽然编程语言支持这么做，但是在不需要使用受查异常的时候还是不用为好。至于在 Diksam 中如何使用，就交给使用者来选择了。

补充知识 异常处理本身的是与非

既然说到了受查异常的是与非，让我们再来看看关于异常处理的两派的论调。

在一本叫作 *Joel on Software*（很有名）的书中，作者 Joel Spolsky 关于异常的论述如下（拙译）：

Exceptions[13]

- **在源代码中很难看到异常。因为不知道哪里会发生异常，所以即使很缜密地检查了代码，还是很难发现其中的错误。**
- **异常赋予了程序多个“出口”。在编写正确的代码中，程序一定能够掌握执行的路径，但是加入了异常后这件事就办不到了。**

Windows 的开发者 Raymond Chen 把异常和早先在返回值中返回错误编码的方式做了比较，请见表 9-1 和表 9-2[14]（拙译）。

表 9-1
错误编码和异常的比较 1

很简单	难	很难
●使用错误编码编写出不好的代码 ●使用异常编写出不好的代码	●使用错误编码编写出良好的代码	●使用异常编写出良好的代码

表 9-2
错误编码和异常的比较 2

很简单	难	很难
●认识到使用错误编码编写出了晦涩难懂的代码 ●使用错误编码编写的，能够分辨出不好的代码和良好的代码	●认识到使用错误编码编写出了通顺易懂的代码	●认识到使用异常编写出了晦涩难懂的代码 ●使用异常编写出来的，能够分辨出不好的代码和良好的代码 ●认识到使用异常编写出了通顺易懂的代码

下面就举一个实际的例子。

```
NotifyIcon CreateNotifyIcon()
{
 NotifyIcon icon = new NotifyIcon();
 icon.Text = "Blah blah blah";
 icon.Visible = true;
 icon.Icon = new Icon(GetType(), "cool.ico");
```

```
7:  return icon;
8: }
```

这段程序本来应该将第 5 行和第 6 行反过来写。这是因为在图标创建失败的时候，第 6 行会发生异常，此时就没有必要把 icon.Visible 设置为 true 了 *。

* 实际试一下的话，在我的环境中没有发生什么特别的问题。

可能上述这个 Windows 编程的经验之谈不太好理解，我再（要举的话还有很多呢）举一个其他的例子。在构建树结构的时候，给父增加子的代码。

```
node.children = new  Node[5];
for (i = 0; i < 5; i++) {
    node.children[i] = new Node();
}
```

在这段代码中，如果前三次循环都正常地执行、第四次的时候发生了异常的话，数组 `node.children` 就会从中间开始变成 `null`。从数据结构上讲，这种情况大多是不被允许的。在这种情况下，即便是在上层的某处捕获到了异常也很难修复这个错误。为了避免这种情况的发生，应该像下面这样编写代码 *。

* 这里假设 `node.children` 的值为 `null`。

```
Node[] nodeArray = new  Node[5];
for (i = 0; i < 5; i++) {
    nodeArray[i] = new Node();
}
node.children = nodeArray;
```

但是，如何能够在程序的所有地方都防止这样的错误发生呢?

结果，话题还是回到是不是应该存在异常处理机制的问题上。当然，完美的异常处理是非常困难的。但是，现实中在对信任关系的要求没有那么严格，使用异常处理还可以准确地将错误传递给上层的情况下，我认为异常处理机制还是有必要的 *。

* 不管怎么说，利用返回值进行手工处理的方式是行不通的。之所以这么说是因为，我不相信人类（当然包括我自己）。

> **对于编写一个将许多数据搜集起来并每日打印一次的小脚本来说，异常可真是个好东西，它可以忽略掉所有会引起问题的地方。我非常喜欢做的就是，利用 `try/catch` 整理程序，并在发生异常的时候将问题通过邮件发给我。但是异常只适合一些粗略的工作或者脚本，并不适合关键性的任务和与维持生命相关的程序。假如在操作系统、核能发电或者心脏手术中使用的高速旋转骨锯的控制软件中使用异常的话，是相当危险的事情。**
>
> ——Joel Spolsky《软件随想录：程序员部落酋长 Joel 谈软件》[15]



9.3 构建脚本

在 3.3.5 节中我们已经接触过了，我认为编程语言的处理器应该尽可能让程序通过一个可执行文件就可以运行。任何一种语言，在其制作的初期都是个不成

熟的语言，因此不可能在一开始安装的时候就让人有所期待。另外，在别人试用语言的时候，要花费很多时间进行高门槛的安装过程可不是一个好主意。

在代码清单 9-7 中例举的异常类程序，在 crowbar 中编写的话就非常简单了，但是要在 C 语言中编写的话我想可能要花不少的功夫。因此，我们需要一种用 crowbar 编写源代码，将 crowbar 本身打包为可执行文件的方法。

另外，虽然 Diksam 中以外部文件的形式保存着现在的 `diksam.lang` 包的源代码，但我还是想要把它打包到可执行文件中去。

在这里让我们研究一下这个方法。

9.3.1 基本思路

首先先来研究一下 crowbar。想法很简单，比如下面这段 crowbar 代码 builtin.crb：

```
function create_exception_class(parent) {
    this = new_object();
    this.parent = parent;
    this.create = closure(message) {
（之后省略）
```

基于上面这段代码，生成了下面的 C 代码。也就是说，crowbar 的代码被作为 C 语言的字符串保存起来。

```
#include <stdio.h>
#include "CRB.h"

static char *st_builtin_src[] = {
    "function create_exception_class(parent) {\n",
    "    this = new_object();\n",
    "    this.parent = parent;\n",
    "    this.create = closure(message) {\n",
（之后省略）
```

之后再进行编译、链接、执行的时候，在加载使用者指定的 crowbar 源代码之前，先编译这段字符串就可以了。

另外，将 builtin.crb 转换成为 C 语言代码的程序是在 crowbar 中编写的。

当然，这段程序必须要在编译 crowbar 的过程中。为了在编译 crowbar 的过程中能让 crowbar 运行起来，我又制作了没有组建构建脚本状态的“minicrowbar”

可执行文件，并且利用 minicrowbar 来完成转换工作。

也不是特意要做这样细致的工作，而是我觉得用 C 语言写这种字符串转换程序不太好。在我的意识中，crowbar 作为一门以 Perl 为目标的语言，确实还是应该在 crowbar 中进行字符串处理。这也是为什么我要尝试着去这么做的原因。

9.3.2 YY_INPUT

保存在 C 语言字符串字面量中的 crowbar 代码，会在使用者编写的程序之前进行编译。

制作 crowbar 的时候，编译采用了 yacc 和 lex 协同作业的方式。因此，最先加载源代码的是 lex（生成的程序）。lex（生成的程序）在默认状态下将从全局变量 yyin 的文件指针中加载源代码（如代码清单 2-2）。

但是，这次不是从文件中加载，而是加载内存中的字符串。在这种情况下，flex 会使用宏 YY_INPUT（但是，这个宏的移植性未必很高）。

像下面这样，通过替换 YY_INPUT 的定义，可将 fle 的标准的输入例程替换为单独的输入例程。

```
/* crowbar.l 的开头 */
#undef YY_INPUT
#define YY_INPUT(buf, result, max_size)(result = my_yyinput(buf, max_size))
```

输入例程要接收缓冲和缓冲大小（fgets() 流）两个参数，缓冲中存入字符串并返回该字符串的字数。缓冲必须以“\0”结尾。

crowbar 的 my_yyinput() 引用了当前解释器的“输入模式”，在有 CRB_FILE_INPUT_MODE 的情况下从文件中输入，在有 CRB_STRING_INPUT_MODE 的情况下从字符串输入。这个“输入模式”保存在 CRB_Interpreter 中。

构建脚本将在创建 CRB_Interpreter 的时候进行编译。在这之后，将使用同一个解释器编译用户程序，此时为了能够让用户程序发生错误时显示正确的行号，解释器中保存的行号会被重置为 1。

9.3.3 Diksam 的构建脚本

在 Diksam 的 book_ver.0.3 中，`diksam.lang` 是以外部文件的形式存在的，单独的可执行文件没办法运行。这对于构建脚本来说的确不太方便。

虽然 Diksam 的程序可以由多个文件组成，但是基本思路还是和 crowbar 一样。

构建脚本并没有在文件系统中保存源文件的实体，因此，会在源文件的搜索路径 `DKM_REQUIRE_SEARCH_PATH`、`DKM_LOAD_SEARCH_PATH` 的开头默认添加上。

另外，和这个修改一并完成的，还有使用者即使不显式 `require` 标准包 `diksam.lang`，它也会被默认 `require` 进来。

9.3.4 三次加载 / 链接

在之前的 Diksam 中，`push_function`、`new` 等指令的操作数是函数和类的索引值，它们通过以下方法取值[1]（在 8.1.5 节中已经介绍过）。

- 在编译的时候，将每个 `DVM_Executable` 中固有的索引值作为操作数。
- 在向 DVM 中加载的时候，将字节码中的对应位置替换为 `DVM_VirtualMachine` 中固有的索引值。

在这种方法中，一个 `DVM_Executable` 被特化到一个 DVM 中，因此在多个 DVM 中不能共享 `DVM_Executable`。

可能有人会问了："当初为什么要创建多个 DVM 呢？"例如，在 Web 应用中，一台服务器上很可能运行着多个应用程序，在这种情况下，不是应该为每个应用程序分别分配不同的 DVM 吗？再举一个其他的例子，在用 Diksam 编写 Diksam 的集成开发环境（IDE）的情况下，驱动 IDE 的 DVM 和在 IDE 的菜单上选择"执行"而启动起来的 DVM，也应该是两个不同的 DVM。

因此，这次我们引入了间接引用对照表，通过它就可以不用再替换字节码中函数和类的索引值了。间接引用对照表保存在 `ExecutableEntry` 中，在加载的

① 在不同的阶段中取值也不同。——译者注

时候创建。

```
struct ExecutableEntry_tag {
    DVM_Executable *executable;
    int         *function_table;    ←函数的间接引用对照表
    int         *class_table;       ←类的间接引用对照表
    int         *enum_table;        ←枚举的间接引用对照表
    int         *constant_table;    ←常量的间接引用对照表
    Static      static_v;
    struct  ExecutableEntry_tag *next;
};
```

并且，按说在加载时局部变量的偏移量也要进行字节码的替换，但是这次并没有修改。这里说的替换通常在同样的机器上运行的 DVM 中，也会出现相同的结果。

9.4 为 crowbar 引入鬼车

crowbar 名字是由“像 Perl 一样的语言”而来的。

近些年来，Perl 被应用在了各个领域中，但是早期的 Perl 只是用来处理文本文件的。说起处理文本文件，最方便的莫过于正则表达式了。

但是想要从零开始制作一个正则表达式引擎的话太困难了，因此我们引入现有的正则表达式程序库“鬼车”。

9.4.1 关于“鬼车”

“鬼车”是小迫先生开发的正则表达式程序库。

> 官方网站（英语）:
> http://www.geocities.jp/kosako3/oniguruma/

在写作本书的时候最新版本为 5.9.1[①]，UNIX（含 Mac 操作系统）和 Windows 都可以安装。许可类型为 BSD 许可，标明著作权、许可条文和免责条款后，可以

① 现在最新版本是 5.9.4。——译者注

在自制软件中（可以不开放源代码）自由使用。

因为使用了这个程序库，所以可以简单地将正则表达式引入到 crowbar 中。在这里要感谢开发者小迫先生。

具体的安装方法和程序库的使用方法，考虑到很有可能会发生变化，请大家直接参考网站的内容。

```
http://avnpc.com/pages/devlang#oniguruma
```

9.4.2 正则表达式常量

在编程语言中处理正则表达式的时候，问题在于要将正则表达式特化到什么程度。

在 Perl 的语言设计上，专门针对正则表达式进行了优化。无论 s///、m/// 还是 s///g 亦或 =~，这些在我看来都太僵化了。对于在 AWK 级别中对文本处理进行特化的语言来说，这样也许很好，但是我不想让 crowbar 变成这样。

反之在 Java 和 PHP 中并没有直接支持正则表达式，而是用程序库的方式支持正则表达式。也许有人会想，这么做也还行吧。在这种情况下，如果只使用单纯的字符串来表现正则表达式的话，在字符串字面量中可能会含有特殊含义的字符，因此不得不进行编码处理。字符串字面量包含的特殊含义的字符，很有可能在正则表达式中也具有特殊的含义。为了使用这样的字符，在正在表达式中还要进行编码，即必须要进行双重编码。因此，例如 Java 中匹配 \ 的正则表达式必须要写成 \\\\。这让人感觉很愚蠢。

不知道是不是为了解决这个问题，在 Python 中引入了 raw string 的概念。在 Python 中，

```
r"字符串"
```

这样在字符串前面加上 r 的话，在这个字符串中像 \ 这样的字符将不再具有特殊含义。如此一来，匹配 \ 的正则表达式只要写成 \\ 就可以了。在 C# 中加上了 @ 的字符串（逐字字符串：verbatim string literal）能达到同样的效果。

但是，如果 \ 没有特殊含义的话，字符串字面量中出现了 " 的时候该怎么办呢？

Python 的使用手册 [16] 中写道：

> 引号可以用反斜杠进行编码，但是这样一来反斜杠本身就成了遗留问题。例如，r"\"" 是正确的字符串字面量，说明由反斜杠和双引号组成了一个字符串，而 r"\" 则是错误的字符串字面量（raw 字符串不能以反斜杠和连续奇数个字符串结尾）。严格地说，（因为反斜杠会编码跟在它后面的引号）**raw 字符串不能以单个反斜杠结束 **。

在我看来，这是一种招致混乱的设计（顺便说一下，C# 的逐字字符串采用了两个双引号写在一起代表一个双引号的设计，这种设计与 Pascal 相似）。

另外，如果想要高效地解释正则表达式，就必须要进行事先编译。但是，对于使用者（crowbar 程序员）来说，每次都编译十分麻烦。在大多数的程序中，正则表达式都不是在运行时组合而成的，因此可以在编译源代码的同时编译正则表达式。这样一来，作为一门编程语言来说，它就需要一种表现“正则表达式字面量”的格式。

在 Ruby 中，通过 %！字符串！的方式可以实现跟 Python 的 raw string 一样的效果。与 Python 不同的是，！可以是任意字符串。这种方法中，只要使用字符串字面量中没有的字符把字符串包起来即可，也很好地回避了 Python 中出现的问题。另外，在 Ruby 中通过 %r！正则表达式！的方式可以表示一个正则表达式字面量，使用这种方式定义的字面量，可以在编译时先编译正则表达式。但在 crowbar 中，% 会被当做模运算符来使用，也充分考虑到了（运算符左右两边不需要输入空格）像 a%r+3 这样的程序。

从而在 crowbar 中，采用了 %%r" 正则表达式 " 的格式。和 Ruby 一样，%%r 后面可以是任意的字符。也就是说，%%r"hoge" 和 %%r!hoge! 两种写法达到的效果是相同的。

但是实际上，这样的语法一旦用多了，就会出现各种各样的符号（用来定义正则表达式），看上去就不那么美观了。

9.4.3 正则表达式的相关函数

crowbar 中与正则表达式相关的函数如下所示。

- reg_match(regexp, subject, region);
  对字符串 subject 使用正则表达式 regexp 进行匹配，如果匹配返回 true，不匹配则返回 false。

可以省略 region，也可以传入一个使用 new_object() 创建的对象。

在 region 中会返回 string、begin、end 三个数组成员。例如正则表达式 "hoge(.*)piyo"，string[0] 中保存着与表达式全部匹配的字符串，string[1] 中保存着与“\1”（也就是正则表达式的 (.*) 部分）匹配的字符串。start、end 返回 string[n] 开始和结束位置 +1（这个设计是照搬鬼车的）。

- reg_replace(regexp, replacement, subject);

  在字符串 subject 中，将匹配正则表达式 regexp 的部分替换为字符串 replacement，并返回替换后的字符串。如果有多个位置匹配，则替换第一个匹配的位置。

- reg_replace_all(regexp, replacement, subject);

  在字符串 subject 中，将匹配正则表达式 regexp 的部分替换为字符串 replacement，并返回替换后的字符串。替换针对所有匹配的位置进行。

  包括 reg_replace() 在内，在 replacement 中可以使用**回溯引用**，replacement 的类型只能是字符串。但我想，给 reg_replace() 传递的 \1 是不是必须要写成 \\1 呢？实际上在 crowbar 中，含有 \n（换行符）和 \t（制表符）的字符串字面量需要特殊处理，因为在 crowbar 中没有将 \025、和 \x5c 之类的八进制或者十六进制的数值嵌入到字符串中（C 语言中有）的功能,所以将 \1 写为 \1 也是没有问题的（看到这肯定会有人问我：“那前面那节的讨论还有什么意义呢？”）。

  ——虽然可能有人会认为以后使用这种设计比较好，但是我认为现在这样也没什么问题。

- reg_split(regexp, subject);

  用 regexp 分割字符串 subject，返回分割后的字符串数组。

9.5 其他

9.5.1 foreach 和迭代器（crowbar）

在 9.1.3 节的注解中写道：“但是，在 crowbar 的标准中，并不是引入 foreach 函数，而是引入 foreach 语法。”如前面所述，有了闭包，即使语言本身不支持，也可以进行类似 foreach 的实现。但在 crowbar 的闭包中，不能使

用 break、continue、while 等语句 *，因此，crowbar 中支持了 foreach。

crowbar 的 foreach 的使用方法如下所示。

```
a = {1, 2, 3, 4, 5, 6};

foreach(v : a) {
    print("(" + v + ")");
}
```

foreach 的语法根据语言而异，比如在 C# 中是这样的。

```
foreach (Object o in hogeCollection)
{
    // 处理
}
```

Java 中虽然没有 foreach，但是对 for 语句进行了扩展，使用方法如下。

```
for (Object o : hogeCollection){
     // 使用 o 进行处理
}
```

crowbar 中的 foreach 结合了上面两种语言的特点。这么做的原因，首先是如果没有 foreach 语句的话，那么程序员之间就没办法在交流的时候说“这里用 foreach 转一下”。可话虽如此，但是像 C# 这样将 in 之类的又短又经常会被用到的单词作为关键字，总觉得有一些不安。

在上面的例子中，使用 foreach 轮询了一个数组，这是因为数组具有迭代器（iterator）的所有方法。

crowbar 中的迭代器采用了 GoF* 的风格，具有以下这些方法。

- first()
  返回迭代器中的第一个元素。
- next()
  将迭代器的指向向后移动一个。
- is_done()
  迭代器移动到超出最后一个元素的位置时返回 true，否则返回 false。
- current_item()
  返回迭代器中当前的元素。

习惯了 Java 的人可能会觉得这样的设计和记忆中的不太一致，这是因为 Java 的迭代器指向数组的元素和元素之间。调用 next() 的时候，迭代器移动到下一

* 如果使用 Java 提出的闭包 unrestricted closure，就可以实现 break 或者 continue。

* GoF 是“Gang of Four”的简称。《面向对象的设计模式》的四位作者组成了四人组，被业界称为“四人帮”。他们的书也被称为 GoF 的书。

个“元素和元素之间”，此时返回的是它跨过的那个元素。

与此相对，crowbar 的迭代器（GoF 风格）是直接指向元素的。因此，使用 current_item() 方法可以取得当前元素，只要不调用 next()，无论调用几次 current_item()，返回的都是同样的元素。

至于哪种设计方式更好，肯定会有各种不同的意见。但是我觉得 Java 设计对于我来说很不好用（只是“看一下”这个元素，迭代器就移动到下一个元素了），因此，这里使用了 GoF 风格的设计方式。

——虽然是这么说，但取得数组的迭代器的方法如果是 GoF 风格的话，本应叫作 CreateIterator()，可这个名字实在是太长了，因此叫作 iterator() 了。只有这点可以说是使用了 Java 的风格，没有与 GoF 风格统一。

数组迭代器的实现如下所示。__create_array_iterator() 是创建迭代器用的隐藏函数，被记录在构建脚本中。数组的 iterator() 方法所返回的迭代器就是调用这个函数取得的。

```
function __create_array_iterator(array) {
    this = new_object();
    index = 0;
    this.first = closure() {
        index = 0;
    };
    this.next = closure() {
        index++;
    };
    this.is_done = closure() {
        return index >= array.size();
    };
    this.current_item = closure() {
        return array[index];
    };
    return this;
}
```

9.5.2 switch case（Diksam）

本节将为 Diksam 引入 switch case。

switch case 语句在 C、Java、C# 等语言中都存在，但是 C 中的 switch case 却很不像话，里面并不能写 break 语句，也就是说，switch case 每次

都要从上至下一直执行到结束。Java 在这点上也是一样。C# 的语法结构看上去和 C 一样，但是如果在 case 的末尾不加上 break 的话就会发生编译错误。这种设计方式正好解决了这个问题 *。此处贯彻了让习惯了 C 和 Java 的程序员容易上手的宗旨，在这点上 Diksam 做了很多妥协。但是在这个问题上如果去迎合 C 语言的话，就不太符合我的审美观了。话说回来，Diksam 中是这样进行 switch case 的。

* 如果在 case 后面没有任何语句（只有 case 语句）的情况下，可以省略 break。

```
switch(a)
case 1 {
    // a 等于 1 时执行
} case 2,3 {
    // a 等于 2 或 3 时执行
} case 4 {
    // a 等于 4 时执行
} default {
    // a 不等于上面的 1、2、3、4 时执行
}
```

并且，Diksam 的 switch case 在原则上只要是能通过 == 进行比较的，就都可以通过 switch 表达式（上例中的 a）和 case 表达式（区别在于在 == 的时候会发生类型转换，switch 的时候不会发生）。因此，字符串等类型也可以使用 switch case。

9.5.3 enum（Diksam）

Diksam 中也同样引入了**枚举类型**（enumerated type）。

```
enum Fruits {
    APPLE,
    ORANGE,
    BANANA
}
```

有了上面的定义，就可以使用 Fruits.APPLE、Fruits.ORANGE、Fruits.BANANA 的**枚举**（enumerator）了。

Diksam 的枚举类型内部保存的是从 0 开始顺序编号的 int，但是并不能作为 int 类型进行四则运算、赋值给 int 类型以及与 int 类型进行比较运算，只能够在同一枚举类之间比较（这些功能可能经常会被用到，因此这里策略性地破坏了

美感，但可以比较大小）。

另外，在用 + 连接左边的字符串时，枚举类型会转换为字符串类型。

```
Fruits f = Fruits.ORANGE;

println("f.." + f); // 输出 "f..ORANGE"
```

如果没有这个设计的话，枚举在编译的时候就可以转换为整数类型了（现在的 Diksam 还不能将字节码保存为文件）。为了将枚举作为字符串输出，必须要把对应的字符串保存在 `DVM_Executable` 中。另外，还必须要和其他源文件进行链接。为了达到这个目的，在 `ExecutableEntry` 中与函数和类一样保存了一份转换对应表（请参考 9.3.4 节）。

9.5.4 delegate（Diksam）

在 Diksam 的语法规则中，函数调用是下面这样的。

```
primary_expression LP argument_list RP
```

也就是说，函数调用表达式是在表达式后面加上括号并且里面括着参数。如果要把语法规则变成下面这样的话，实际上简单了不少。

```
IDENTIFIER LP argument_list RP
```

如果变成了上面这样，当然，是为了实现在类似于 C 的语言中所说的函数指针。例如为 GUI 的按钮分配处理的时候，在 Java 中要创建一个实现了特定接口的类的实例（事件监听器），并将它设置到按钮中。这种方法存在以下的问题：

- 需要为此特意去定义一个类，不仅麻烦也会使代码变得冗长。
- 按下按钮时，处理会被编写在别的类里面，对于这个类来说等于放宽了类的封装。虽然使用内部类可以解决这个问题，但也因此又带来了内部类的使用问题。（对于实现语言的人来说）这个方法太麻烦了，而且对于初学者来说也不太容易掌握。
- 在按下按钮的时候，事件也可以由承载了按钮的类（`JFrame` 等）接收。如果使用了这种处理方式，在这个类中有两个按钮的话，就无法为它们分配单独的动作。

因为只是“想要执行按下按钮时的处理”，所以很自然地就会想到，如果能只登录描述了处理的函数就好了（可能不得不使用闭包，但是在 Diksam 中却不能使用）。

为了能够达到上述效果，就需要为静态类型的语言 Diksam 引入“函数类型”（在 C 语言中的话就是指向函数的指针型）。

如果想要声明一个“接收 int 参数，返回 double 函数”的类型，肯定不能像 C 语言这样编写代码。

```
double (*func)(int);
```

说到 Java，从 Java7 开始就有了要引入闭包的说法，当初设想的是下面这样的代码。

```
double(int) func;
```

但是在 Java（Diksam）中存在检查异常，如果把 throws 也作为方法必要的信息，就要写成下面这样。

```
double(int) throws HogeException, PiyoException func;
```

上面的写法是因为在语法上存在不确定性，所以需要改为下面这样的写法。

```
double(int) throws HogeException | PiyoException func;
```

相反也可以试着像下面这样定义。

```
{ int => double } func;
```

从 2009 年 5 月到现在，就连是否要引入（闭包）这个问题本身都还没有得出结论 *。说句题外话，无论哪种写法我都觉得太长了（使用起来至少也要像 C 语言中的 typeof 那样）。

* 很明显，在 Java7 中并没有加入这个功能。

因此，在 Diksam 中，引入了 C# 风格的关键字 delegate。

```
delegate double Func(int value) throws HogeException, PiyoException;
```

根据这个描述，Func 被定义为“接受 int 型参数，返回 double，可能会抛出 HogeException 和 PiyoException 的函数”的类型。

基于上面的定义，就可以声明 Func 类型的变量，参数中也可以接收 Func 类型了。

```
// 定义 Func
delegate double Func(int value) throws HogeException, PiyoException;

// 定义函数
double func(int value) throws HogeException, PiyoException{
  ⋮
}
```

```
// 将 func 赋值给 Func 型的变量 f
Func f = func;

// 通过 f 调用 func
f(5);
```

与 C# 的 delegate 不同，Diksam 的 delegate 类型并不是类的对象。因此，没有必要使用 new*，也可以说不能 new。另外，不能将多个函数赋值到一个 delegate 中。

* C# 从 2.0 开始也不需要 new 了。

另外，在给 delegate 类型的变量赋值的时候，和方法重写时一样，返回值必须共变，参数必须反变。

方法也可以赋值给 delegate 类型的变量，此时，方法所在对象的引用也会被保存到变量中。因此，方法在作为事件句柄被调用的时候，也可以和平常一样使用 this 引用。

在实现上，delegate 类型的值作为 DVM_Object 的联合体之一保存在堆中。delegate 型变量如果保存的是方法的话，就必须要同时保存方法所在对象的引用，因此，需要保存的信息如下所示。

```
/* 保存 delegate 信息的结构体 */
typedef struct {
    /* 如果保存的是方法，那么这个成员保存的是方法所在对象的引用。如果是函数则为 null */
    DVM_ObjectRef       object;
    /* 函数或者方法的索引值 */
    int                 index;
} Delegate;

/* 在 DVM_Object 结构体中通过联合体保存上述结构体 */
struct DVM_Object_tag {
    ObjectType type;
    unsigned int        marked:1;
    union {
        DVM_String      string;
        DVM_Array       array;
        DVM_ClassObject class_object;
        NativePointer   native_pointer;
        Delegate        delegate;    ←这个
    } u;
    struct DVM_Object_tag *prev;
    struct DVM_Object_tag *next;
};
```

那么，关于 delegate 对象的创建时机，从语言实现的角度讲，无论是函数还是方法，让调用总是通过 delegate 对象的话比较容易实现。无论是调用 print("hello"); 这样的函数，还是调用 obj.method() 这样的方法，在它们对 print、obj.method[1] 进行计算的时候就会创建 delegate。但是，向堆中保存对象是一项开销很大的处理，大多数情况下，print 也不会赋值给其他变量，而是立即调用。为了不因个别情况而影响整体的效率，delegate 对象在以下时机被创建。

> 1. 在函数的情况下，当函数赋值给delegate变量的时候。
> 2. 在方法的情况下，通过非立即调用的形式进行了（表达式）计算的时候。

另外，delegate 对象在指向方法的时候也引用了对应的对象，这样做会导致这个对象不能成为 GC 的目标。因此也需要对 GC 进行修改。

9.5.5 final、const（Diksam）

crowbar 想要定义常量的时候，要使用 final（Java 风格）。

```
final HOGE = 10;
```

在变量声明时如果加了 final 的话，在赋了初始值之后，就不能再对其进行赋值了。并且，必须要在声明的同时完成赋初始值的动作。final 也可以用于局部变量。

在 Diksam 中，同样使用了 final。

```
final int HOGE = 10;
```

在 Diksam 中函数的形式参数、catch 子句中接收异常的变量，都默认为是 final 的。这样做的目的在于，不让从被调用的位置或者异常发生的源头获得的重要信息被稀里糊涂地覆盖掉。

在 Diksam 中允许分割编译，并且不存在跨源文件的全局变量。为了不出现乱用全局变量的情况，我认为最好的方式是通过 get_xxx() 和 set_xxx() 函数。但是在大规模的程序中，可能需要被所有程序引用的全局常量。

但是，我认为，与其抛弃“不存在跨源文件的全局变量”等（和其他语言相

① 这两行代码在被（）调用前，会被当做表达式。——译者注

比有些特别）的设计，直接允许使用全局变量，不如声明加了 final 的全局常量可能更好一些。但实际上并没有这么简单。Diksam 具有顶层结构，顶层结构是由被执行的语句组成的。因此，即使在函数外声明了变量（如果是 C 语言的话，就是声明全局变量），在声明语句被执行前是得不到初始值的。这样一来，因为 require 了的其他文件源代码的顶层结构（在编译时）是绝对不会执行的，所以即使用 final 声明的常量可以被其他源文件 require。但是在编译时来看，它并没有被赋值，因此达不到预期的效果。

因此，在 Diksam 中配合着 final 引入了 const 这个关键字。const 的使用方法如下所示。

```
const HOGE = 10;
```

因为通过初始值可以判断类型，所以 const 无需指定类型。

const 与函数定义、枚举、delegate 的声明一样，不能写在函数内。另外，可以从其他源文件中被引用。

在为 const 指定的常量设置值的时候，需要考虑以下的情况。

1. 与C语言的预处理相同，在编译前就要置换常量。
2. 在编译时置换常量。
3. 编译时与变量采取同样的实例，在开始执行时再将值代入。

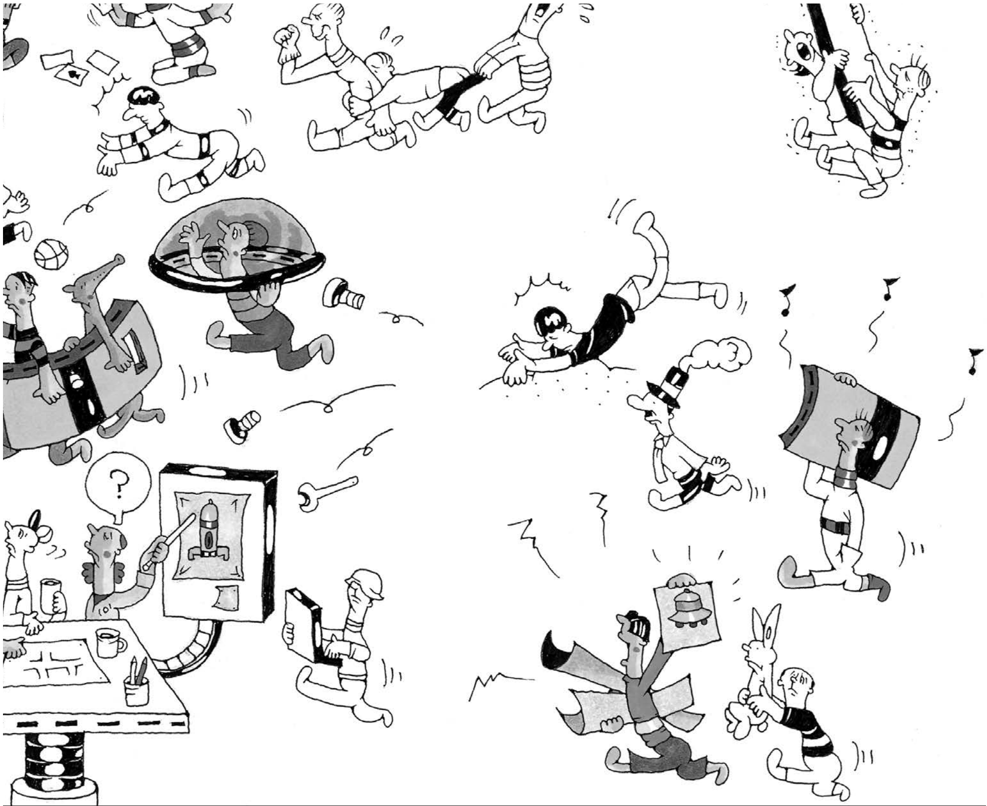
1、2 在编译前和编译时进行替换的方法在执行效率方面具有优势，但是对于使用者来说，会由于不能定义如下形式的常量而困扰。

```
const HOURS_IN_DAY = 24;// 一天 24 小时
const MINUTES_IN_DAY = HOURS_IN_DAY * 60;// 1 天 = 24 × 60 分
```

如果要在编译时或者编译前置换常量，那么这些表达式在编译时就必须要全部执行。如“数组大小”等被作为常量表达式处理的话，就需要（在编译时）调用数组的 size() 方法。这样的做法太困难了，因此 Diksam 还是采用了方法 3。

所有包含有代码的 const 常量的初始值都将作为字节码保存在 DVM_Executable 中，在编译 / 加载源代码的时候再去执行它们。初始值中可以书写任意的表达式，既可以使用 new 分配对象，也可以使用原生函数分配特定的系统资源（例如 stdin、stdout、stderr 文件指针等）。

因为 const 常量会被其他文件链接，所以与函数、类、枚举一样在 ExecutableEntry 中保存了转换对应表。



附录



附录A crowbar 语言的设计

本章将要说明在本书中 crowbar 的最终版本（book_ver.0.4）的设计。但是，本章中的内容既不是定稿版的文档，也不是很严谨。因为如果想要严谨地描述一门语言的设计需要相当的篇幅。

A.1 程序结构

构成程序的要素

crowbar 的程序由以下要素构成。

1. 顶层结构

顶层结构（toplevel）由**语句**（statement）组成。crowbar 的程序由处理器指定的源文件的顶层结构开始执行。

2. 函数定义

函数定义(function definition ）是定义可以（可能）从其他位置调用的函数。在顶层结构的语句按顺序执行的时候会忽略函数定义。因为函数允许回溯引用，所以其定义的位置既可以在调用的位置之前，也可以在调用的位置之后。

A.2 文字语法规则

A.2.1 源文件的编码

crowbar 的源文件使用与处理器相同的字符编码。当前确定的是在 UNIX 环境中为 EUC 和 UTF-8，在 Windows 环境中为 GB2312。除了写注释和定义字符串

字面量之外的情况，都不能使用非 ASCII 字符。

A.2.2 关键字

下面这些单词作为 crowbar 的关键字，不能作为变量名、函数名使用。

```
break catch closure continue else elseif false final finally for
foreach function global if null return throw true try while
```

A.2.3 空白字符的处理

空格（ASCII 码的 0x20）、制表符和换行符在源文件中除了区分不同的标识符外没有其他含义。

A.2.4 注释

在 crowbar 中从 # 开始到本行结束都会被作为注释。

A.2.5 标识符

被当做变量名、函数名、类名等使用的标识符，需要遵从以下规则。

- 第一个字符必须是英文字母（A~Z，a~z）或者是下划线。
- 从第二个字符开始可以是英文字母、下划线或数字（0~9）。

crowbar 的标识符中不允许使用中文等 ASCII 字符集中不包含的字符。

A.2.6 字面量

1. 真假值字面量

真假值字面量只有 `true`（真）和 `false`（假）。



2. 整数字面量

整数字面量由 `0` 或者“`1~9` 后面跟着 0 个或以上的 `0~9`”组成。

目前为止还不支持八进制和十六进制的表示方式。

3. 实数字面量

实数字面量由“1 个以上 `0~9` 的组合、点、1 个以上 `0~9` 的组合”组成。

`.5` 或者 `1.` 都不是实数字面量。

4. 字符串字面量

字符串字面量是由双引号括起来的字符串。在字符串字面量中 `\n` 表示换行，`\t` 表示制表符，`\\` 表示 `\`。

5. null 字面量

`null` 字面量用来表示引用类型没有指向任何值。

6. 数组字面量

在花括号 {} 中将元素用逗号分开，并初始化元素就创建了数组字面量。数组字面量的详细内容请参考 A.3.3 节。

在最后一个元素的后面有没有逗号都可以。

7. 正则表达式字面量

由 `%%r` 和任意一个将正则表达式括起来的字符组成正则表达式字面量。

【例】

```
%%r"[a-z]*"
%%r!hoge(.)a\1! ←用括号括起来的部分可以在 \1 的位置被回溯引用
```

A.2.7 运算符

以下这些字符（也许只是一部分）会被当做**运算符**（operator）解释。

```
, && || = == != > >= < <= + - * / %
+= -= *= /=%= ++ -- ! . () []
```

A.2.8 分隔符

以下的字符会被解释为**分隔符**(punctuator)。

```
() {} ; : ,
```

A.3 数据类型

A.3.1 类型一览

在 crowbar 中存在以下**类型**(type)。

逻辑类型：`true` 或者 `false`。

整数类型：能够表达的范围与处理器编译的 C 语言环境中的 `int` 类型一致。

实数类型：能够表达的范围与处理器编译的 C 语言环境中的 `double` 类型一致。

字符串类型：字符串类型的详细内容请参考 A.3.2 节。

数组类型：数组类型的详细内容请参考 A.3.3 节。

对象类型：对象类型的详细内容请参考 A.3.4 节。

函数类型：函数类型的详细内容请参考 A.3.5 节。

原生指针类型：原生指针类型的详细内容请参考 A.3.6 节。

A.3.2 字符串类型

crowbar 的字符串类型属于引用类型。但是因为字符串本身不能改变（ immutable ），所以使用者没有必要意识到它是一个引用类型。

crowbar 的字符串类型的内部表现形式为，处理器编译的 C 语言环境中的宽字符串。

字符串具有以下的**模拟方法**(fake method)，通过方法调用的形式取得字符串的相应信息。

使用范例	含义
`len = s.length();`	取得字符串的长度
`s2 = s.substr(3, 5);`	截取字符串，并返回包含截取内容的新字符串。第 1 个参数指定截取开始的位置（第 1 个字符是 0），第 2 个参数指定要截取的长度

A.3.3 数组类型

crowbar 是非静态类型的语言，因此数组中可以保存所有的类型，不仅如此，各种数据类型可以同时被装在一个数组里面。

crowbar 的数组类型可以通过数组字面量和 `new_array()` 函数创建。

```
# 创建以整数、字符串、实数、数组为元素的数组
a1 = {1, "abc", 10.0, {1, 2, 3}};

# 创建有 10 个元素的数组（元素的初始值为 null）
a2 = new_array(10);
```

取得数组元素和为元素赋值都需要使用 `[]` 运算符。数组下标从 0 开始。

```
# 取得数组的元素
print("a[3].." + a[3]);

# 为数组元素赋值
a[5] = 10;
```

crowbar 的数组是引用类型。

数组具有以下的模拟方法，通过方法调用的形式操作数组。

使用范例	含义
`a.add(3);`	向数组的末尾添加元素
`size = a.size();`	取得数组的元素个数
`a.resize(new_size);`	改变数组的元素个数。如果新指定的大小小于当前数组的元素个数，则根据新的大小舍弃数组后面的元素。如果大于当前元素个数，则增加元素并把这些元素设置为 `null`
`a.insert(2, 3);`	在第 1 个参数指定的下标的元素前，插入第 2 个参数中指定的值
`a.remove(3);`	删除参数指定的下标的元素，并将后面的元素向前移动一个下标。结果是数组的元素个数减少 1
`ite = a.iterator();`	取得数组的迭代器。关于迭代器请参考 A.3.7 节

A.3.4 对象类型

对象类型是由多个**成员**组成的类型。数组通过下标指定元素，对象则是通过成员名称指定。

对象通过 `new_object()` 函数创建。创建的对象通过对指定的成员名称赋值的形式，为对象添加成员。

```
# 创建空的对象
o = new_object();

# 添加对象的成员
o.x = 10;
o.y = 20;

# 引用对象的成员
print("o.x.." + o.x + ", o.y.." + o.y + "\n");
```

对象类型是引用类型。

A.3.5 函数类型

不仅可以通过函数名赋值给其他变量，还可以通过 () 进行调用。

```
function a() {
    print("hello!\n");
}

c = a; # 将函数 a 代入 c
c();  # ←输出 "hello!"
```

可以使用关键字 `closure` 在表达式中定义一个无名的函数。这种方式被称为**闭包**（closure）。

```
c = closure() {
    print("hello!\n");
};
c();  ←输出 "hello!"
```

从闭包内部可以引用到创建闭包所在位置的局部变量。即使在闭包调用的时候，创建闭包的函数已经结束的情况下也是一样。

A.3.6　原生指针类型

原生指针类型是保存在原生函数内部使用的值（如文件指针，`FILE*`）的类型。在原生函数外的原生指针类型只能够进行复制和等值比较。

现在的原生指针类型只在文件指针和正则表达式中使用。

原生指针类型可以由原生函数定义**终结器**（finalizer）。终结器是在GC要释放原生指针类型所指向的对象时执行的函数。但是，因为终结器的执行时机不能被应用程序预测，所以有关重要资源的释放不应该依赖终结器。

A.3.7　迭代器

迭代器（iterator）是表现循环的类型。

迭代器是一个设置了一些闭包方法（method）的对象，并不是处理器中定义的特殊类型。因此，在用户程序中也可以定义迭代器。

迭代器是具有以下方法的对象。

`first()`：返回迭代器中的第一个元素。

`next()`：将迭代器游标向后移动一个。

`is_done()`：迭代器移动到超出最后一个元素的位置时返回`true`，否则返回`false`。

`current_item()`：返回迭代器中当前的元素。

crowbar的数组可以通过`iterator()`方法得到数组的迭代器。另外，在使用`foreach`语法的时候，就是基于迭代器循环的。

A.3.8　异常

在crowbar中，异常是被设置了一些闭包作为方法的对象。可以使用原生函数`new_exception()`来创建异常。

使用者在创建“异常”的时候，需要使用内建函数`create_exception_class()`。这个函数以“父类”为参数创建一个新的异常类。通过调用这个类的`create()`方法，创建属于这个类的实例。

```
# 在构建脚本中对 ArithmeticException 定义的描述
ArithmeticException = create_exception_class(RuntimeException);

# 通过 create() 方法生成实例。
e = ArithmeticException.create();
```

异常实例中的 `child_of()` 方法用于检查异常的类型。在上面的例子中，ArithmeticException 是 RuntimeException 的子类，e 是 ArithmeticException 的实例，因此如果调用 `e.child_of(RuntimeException)` 的话，传入上面两个类都会返回 `true`。

A.4 表达式

所谓**表达式**(expression)，就是通过运算符将字面量或者标识符关联起来。

A.4.1 类型转换

在 crowbar 中使用双目运算符（+、-、*、/、%）和比较运算符（==、!=、>、>=、<、<=）时，如果两边的类型不同的话，crowbar 会基于下面的规则对类型进行扩展。

1. 不论左边还是右边，只要其中一边是整数，另外一边是实数的时候，整数会转换为实数。
2. 在左边是字符串右边是逻辑类型、整数类型或者实数类型的时候，右边会转换为字符串。

上述类型转换，在算术赋值运算符（+=、-=、*=、/=、%=）的情况下同样适用。

A.4.2 运算符一览

crowbar 的运算符一览如下所示（优先顺序由高至低）。

运算符	含义
++ -- () [] .	自增、自减、函数调用、引用数组元素、引用对象的成员

（续）

<table>
<tr><th>运算符</th><th>含义</th></tr>
<tr><td>（单目）- !</td><td>符号反转、逻辑非</td></tr>
<tr><td>* / %</td><td>乘法、除法、模</td></tr>
<tr><td>+ -</td><td>加法、减法。加法运算符可以连接字符串</td></tr>
<tr><td>< <= > >=</td><td>比较大小。字符串也可以比较大小（设计以 C 语言的 strcmp() 为基准）</td></tr>
<tr><td>== !=</td><td>等值比较。字符串也可以进行比较（比较的不是引用而是值）</td></tr>
<tr><td>&&</td><td>逻辑与（AND）运算符。短路运算符在表达式 a && b 中，如果 a 为假的话，b 就不做判断了</td></tr>
<tr><td>||</td><td>逻辑或（OR）运算符。短路运算符在表达式 a || b 中，如果 a 为真的话，b 就不做判断了</td></tr>
<tr><td>= += -= *= /= %=</td><td>赋值运算符。在 crowbar 中会通过最初的赋值创建变量。另外，如果在赋值表达式前面加上 final 的话，就会变成不能再次赋值的常量定义</td></tr>
<tr><td>,</td><td>逗号运算符。以从左到右的顺序计算表达式，返回右边表达式的值</td></tr>
</table>

在 crowbar 中，只存在后置的自增和自减运算符。并且，其动作与 C 等语言中的前置运算符效果相同（不会等到最后，而是立刻自增表达式的值）。



A.5 语句

A.5.1 表达式语句

所谓**表达式语句**（expression statement），就是在表达式后面加上分号。

```
# 调用 print() 函数的表达式，后面加上分号
print("hello,world\n");

# 自增表达式后面加上分号
i++;

# 无副作用的表达式也可以成为表达式语句，但没有任何意义
5;
```



A.5.2 if 语句

if 语句是根据条件进行判断并进行分支处理的语句。

```
if （条件表达式 1） {
    # 条件表达式 1 为真时执行的处理。
} elseif （条件表达式 2） {
    # 条件表达式 1 不为真，且条件表达式 2 为真时执行的处理。
} else {
    # 所有条件表达式都不为真时执行的处理。
}
```

elsif 子句和 else 子句都是可以省略的。另外，可以编写任意多个 elsif 子句。

与 C 等语言不同的是，即使只包含一行语句，也不能省略花括号（{}）。

A.5.3 while 语句

while 语句是进行循环操作的语句。

```
标识符 :
while （条件表达式） {
    # 在条件表达式为真的情况下，此处的代码会反复执行。
}
```

“标识符 :”的部分被称为**标签**（label）。标签可以省略。

与 if 语句相同，即使只包含一行语句，也不能省略花括号（{}）。

A.5.4 for 语句

for 语句是进行循环操作的语句。

```
标识符 :
for（第 1 表达式 ； 第 2 表达式 ； 第 3 表达式）{
    # 在第 2 表达式为真的情况下，此处的代码会反复执行。
}
```

“标识符 :”的部分被称为**标签**（label）。标签可以省略。

让我们先看一段 for 语句的代码：

```
# 创建一个数组
a = new_array(10);

# 循环取得数组的每个元素
for(i = 0; i < a.size(); i++){
   # 输出每个元素
   print("a[" + i + "].." + a[i]);
}
```

基于上面这段代码，其中的第 1 表达式、第 2 表达式、第 3 表达式也有其他的叫法。

第 1 表达式通常的作用是对判断循环是否需要继续的变量进行初始化，因此也叫作初始化表达式。

第 2 表达式通常的作用是根据初始化表达式中创建的变量，以此判断是否需要继续循环，因此也叫作判断表达式。

第 3 表达式通常的作用是重新设置循环的判断条件（一般都是 ++ 或者 --），因此也叫作增量表达式。

上述 for 语句中，除了在 continue 时会执行第 3 表达式之外，其他与下面的 while 语句效果相同。

```
第 1 表达式 ;
while （第 2 表达式） {
    # 语句
    第 3 表达式 ;
}
```

第 1 表达式、第 2 表达式、第 3 表达式都是可以省略的。在省略了第 2 表达式时意味着永远返回真。

A.5.5 foreach 语句

foreach 语句是利用迭代器进行循环操作的语句。

```
标识符 :
foreach （变量 : 集合） {
    # 语句
}
```

“标识符 :”的部分被称为**标签**（label）。标签可以省略。上述 foreach 语句

等同于下面的 for 语句。

```
for (ite = 集合 .iterator(); !ite.is_done(); ite.next()) {
    变量 = ite.current_item();
    # 语句
}
```

但是，实际上在 foreach 语句中引用不到迭代器，在上述例子中变量 ite 并不存在。

A.5.6 return 语句

return 语句是从函数中跳出的语句。

```
return 表达式 ;
```

在省略表达式的时候，该函数返回 null。

A.5.7 break 语句

break 语句是用于跳出循环的语句。

```
break 标识符 ;
```

标识符可以省略。在省略的情况下，跳出最内侧的循环。

在指定了标识符的情况下，跳出持有同样标识符的循环。

A.5.8 continue 语句

continue 语句用于跳转到循环的末尾。

```
continue 标识符 ;
```

标识符可以省略。在省略的情况下，以最内侧的循环为对象。

在指定了标识符的情况下，以持有同样标识符的循环为对象。

在以 for 为对象进行 continue 的时候，continue 后会紧接着计算 for 语句的第 3 表达式。

A.5.9 try 语句

try 语句是用于执行异常处理的语句。

```
try{
    # 语句
} catch （变量名） {
    # 语句
} finally {
    # 语句
}
```

在 try 子句中如果发生了异常就会执行 catch 子句。发生的异常会被设置到 catch 子句声明的变量中。catch 子句和 finally 子句只要存在其中的任何一个，另外一个就可以被省略。

无论异常是否发生，无论是否存在 catch 子句，finally 子句都一定会执行。在 try 子句或者 catch 子句中，如果执行了 break、continue、return 来中断处理，那么也会执行 finally 子句。如果使用 break、continue、return 来中断 finally 子句的话，相比 finally 的执行结果，try 子句或者 catch 子句中语句的执行结果会更优先。例如，try 子句内执行了 return 5;，在返回前执行了 finally 子句中的 return 3;，在这种情况下，最终被作为返回值返回的是 5。

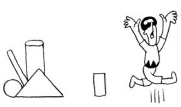
A.5.10 throw 语句

throw 语句是用于抛出异常的语句。

```
throw e;
```

e 在通常情况下会使用通过 new_exception() 创建的异常对象，但是整数类型等其他所有类型都可以被 throw。

在 try 子句内发生的异常，（如果有的话）会在 catch 子句中被捕捉。当 catch 子句不存在或者异常发生在 try 语句之外的时候，会终止当前正在执行的函数并返回到调用者（或者叫上一层）的处理中。如果调用者也没有 catch 这个异常的话，则会沿着调用层级回溯，直到顶层结构。如果顶层结构中也没有捕捉这个异常，就会输出**栈轨迹**（stack trace）并终止处理。



crowbar 中的栈轨迹在调用 new_exception() 的时候就被创建了。

A.5.11 global 语句

global 语句是为了在函数内引用全局变量而使用的语句。在 crowbar 中，如果不使用 global 语句进行声明的话，在函数内就访问不到全局变量。

```
# 引用了 STDIN，STDOUT 两个全局变量
global STDIN,STDOUT;
```

A.6 函数

A.6.1 函数定义

函数使用以下形式定义。

```
function 函数名 （参数） {
    # 语句
}
```

请参考下面的示例。

```
# 接收两个参数并返回它们的和的函数
function sum (a, b) {
    return a + b;
}
```

A.6.2 局部变量

在函数内声明的变量就是**局部变量**（local variable）。

局部变量的作用域只在当前函数内。与 C 语言不同的是，函数内的程序块并不会形成作用域[①]

① 也就是说，局部变量的作用域是当前函数，而不是当前程序块。——译者注

附录 B Diksam 语言的设计

以下将要介绍的是在本书中 Diksam 最终版本（book_ver.0.4）的设计。与附录 A 一样，内容并不是很严谨。

B.1 程序结构

B.1.1 构成程序的要素

Diksam 的程序由以下要素构成。

1. 声明部分

声明部分由 `require` 声明和 `rename` 声明组成。声明部分必须在源文件的开头，但也可以省略。

2. 顶层结构

顶层结构（toplevel）由**语句**（statement）组成。Diksam 的程序从处理器中指定的源文件的顶层结构开始执行。

3. 函数定义或声明

函数定义（function definition）定义可以（可能）从其他位置调用的函数。在顶层结构的语句顺序执行时会忽略函数定义。因为函数允许回溯引用，所以其定义的位置既可以在调用的位置之前，也可以在调用的位置之后。

函数声明（function declaration）只声明了函数的**签名**（signature）。对于函数声明来说，必须存在与其对应的定义。

4. 类型定义

类型定义包括**类定义**（class definition）、**接口定义**（interface definition）、**枚举类型定义**（enumerated type definition）和**delegate 类型定义**（delegate type definition）。

类型定义是为了定义可能在其他位置使用的数据类型。与函数一样，会在顶

层结构的语句执行时被忽略，并且允许回溯引用。

下面是几个示例。

仅由顶层结构组成的 Diksam 程序：

```
println("hello, world.");
```

开头含有声明部分的 Diksam 程序：

```
// 将标准函数 println 改名为 print_line
rename diksam.lang.println print_line;

print_line("hello, world.");
```

包含函数定义的 Diksam 程序：

```
// 函数定义
void func() {
    println("hello, world.");
}

// 调用定义的函数
func();
```

包含类定义的 Diksam 程序：

```
public class Point {
    private double x;
    private double y;
    void print() {
        println("x.." + this.x + ", y.." + this.y);
    }
    constructor initialize(double x, double y){
        this.x = x;
        this.y = y;
    }
}

// 创建 Point 类的实例
Point p = new Point(10, 20);
// 输出 p 的内容
p.print();
```

B.1.2　分割编译

通过声明部分的 `require` 声明指定的**包**（package），达到能够使用在其他

源文件中定义的函数和类的目的。

在 Diksam 中，包由一个源文件（扩展名 .dkh）或者两个源文件（扩展名 .dkh 和 .dkm）的组合组成。

下面的例子通过 require 了 hoge.piyo.fuga 包，达到了可以使用 fuga.dkh 中定义的类和函数的目的。

```
require hoge.piyo.fuga;
```

在 .dkh 文件中描述了只有在函数的签名声明的情况下，实际调用函数的时候处理器才会对 fuga.dkh 对应的实现文件 fuga.dkm 进行搜索并编译 / 加载。这种机制被称为**动态加载**（dynamic load）。

被 require 文件的搜索路径，即环境变量 DKM_REQUIRE_SEARCH_PATH，在 UNIX 环境中使用逗号分隔，在 Windows 环境中使用分号分隔。因为 Diksam 的包名和文件路径的层级是一致的，所以只需要指定搜索路径就可以以此为起点，通过包名的层级搜索文件了。

在没有设定 DKM_REQUIRE_SEARCH_PATH 的情况下，当前目录将成为唯一的搜索路径。

另外，动态加载时的搜索路径可以通过环境变量 DKM_LOAD_SEARCH_PATH 取得。

diksam.lang 包已经默认为 require，因此使用者无需自己再进行 require。

被 require 的源文件中，顶层结构将被忽略，不会执行。

B.1.3 解决命名冲突

在 require 了多个包的时候，很可能会发生类名、函数名的命名冲突。在这个时候，可以通过声明部分的 rename 声明为标识符改名，以此方法来回避命名冲突。

下面是将 diksam.lang 包下面的 print 函数改名为 myprint 的例子。

```
rename diksam.lang.print myprint;
```

B.1.4 关于全局变量的链接

在顶层结构中声明的变量（全局变量）相对于源文件独立的，不能进行链接。如果想要让某个包的全局变量被其他的包访问的话，就必须要声明 get_xxx()、set_xxx() 这样的函数。

B.2 语法规则

B.2.1 源文件的字符编码

Diksam 的源文件使用与处理器相同的字符编码。当前确定的是在 UNIX 环境中为 EUC 和 UTF-8，在 Windows 环境中为 GB2312。除了写注释和定义字符串字面量，其他情况下不能使用非 ASCII 字符。

B.2.2 关键字

下面这些单词作为 Diksam 的关键字，不能作为变量名、函数名、类名等使用。

```
abstract boolean break case catch class const constructor  continue
default delegate do double else elsif enum false final finally for
foreach if instanceof int interface native_pointer new null override
private public rename require return string super switch this throw
throws true try virtual void while
```

（注）foreach 是为将来准备的关键字，现在并没有使用。

B.2.3 空白字符的处理

空格（ASCII 码的 0x20）、制表符和换行符在源文件中除了区分不同的标识

符外没有其他含义。

B.2.4 注释

在 Diksam 中可以使用以下两种注释方式。

- 以 /* 开始以 */ 结束的注释，这种注释不能嵌套使用。
- 以 // 开始直到本行结束的注释

B.2.5 标识符

被当做变量名、函数名、类名等使用的标识符，需要遵从下面的规则。

- 第一个字符必须是英文字母（A~Z，a~z）或者是下划线。
- 从第二个文字开始可以是英文字母、下划线或数字（0~9）。

Diksam 的标识符中不允许使用中文等未被 ASCII 字符集涵盖的字符。

B.2.6 字面量

1. 真假值字面量

真假值字面量只有 true（真）和 false（假）。

2. 整数字面量

整数字面量由“0”或者“1~9 后面跟着 0 个或以上的 0~9”组成。

如果在前面加上“0x”或者“0X”前缀的话，可以表示十六进制整数。

目前为止还不支持八进制表示方式。

3. 实数字面量

实数字面量由“1 个以上的 0~9 的组合、点、1 个以上的 0~9 的组合”组成。

.5 或者 1. 都不是实数字面量。

4. 字符串字面量

字符串字面量是由双引号括起来的字符串。在字符串字面量中 \n 表示换

行，\t 表示制表符，\\ 表示 \。

5. null 字面量

null 字面量用来表示引用类型没有指向任何值。

6. 数组字面量

在花括号 {} 中将元素用逗号分开并对元素进行初始化就创建了数组字面量。数组类型为该字面量中第一个元素的类型的数组。最后一个元素的后面有没有逗号都可以。

B.3 数据类型

B.3.1 基础类型

Diksam 存在以下**基础类型**。

1. void类型（void）

void 类型表示函数是没有返回值的特殊类型。只能作为函数或者方法的返回值使用。

2. 逻辑类型（boolean）

可以是 true 或者 false。

3. 整数类型（int）

表示整数的类型。能够表达的范围与编译器以及编译 VM 的 C 语言环境的 int 类型一致。

4. 实数类型（double）

表示实数的类型。能够表达的范围与编译器以及编译 VM 的 C 语言环境的 double 类型一致。

5. 字符串类型（string）

表示字符串的类型。其内部表现形式为编译器以及编译 VM 时 C 语言环

境下的宽字符串。字符串类型属于引用类型。但是，因为字符串本身不能改变（immutable），所以使用者没有必要意识到它是一个引用类型。

B.3.2 类 / 接口

类和接口都属于用户自定义类型。

关于类定义和接口定义的详细内容请参考 B.7 节。

类和接口都是引用类型。

B.3.3 派生类

派生类型是由基础类型、类、接口派生出来的类型。

存在以下两种派生类型。

1. 数组类型

Diksam 的数组类型，其元素数无须在声明时决定，是可以动态创建的引用类型。

2. 函数类型

函数和方法都是函数类型。函数类型不存在变量，并可以赋值给适当的 `delegate` 类型。

数组类型可以使用递归。

`int`类型数组的数组：

```
int[][] a;
```

函数类型通过 `delegate` 类型可以实现“函数的数组”、“返回函数的函数”等。

B.3.4 枚举类型

枚举类型通过下面的方式定义。在最后一个元素后面有没有逗号都可以。

```
enum 枚举类型名称 {
```

```
    枚举名称,
    枚举名称,
    枚举名称,
  ⋮
}
```

枚举类型的值（枚举）如下所示，使用枚举类型名称 . 枚举名称的形式表示。

```
Fruits.ORANGE
```

枚举类型在参与加法运算（使用 + 运算符）时，如果左边是字符串（枚举类型在右边）的话，枚举类型会转换为字符串。

枚举类型可以使用比较运算符（==、!=、>、>=、<、<=）和 switch 判断分支。

B.3.5 delegate 类型

delegate 类型是指向函数的引用类型。通过在关键字 delegate 后面加上与函数签名相同的形式定义。

下面的实例以一个 Window、两个 int 以及一个 MouseButton 枚举为参数，返回值为 void 的 delegate 类型（在后述 Windows 版的 Diksam 中会被实际使用到）。

```
// 在 Window 按下鼠标的事件
delegate void MouseButtonDownProc(Window window, int x, int y,
                                  MouseButton button);
```

通过上述类型定义，就可以像下面这样声明 delegate 类型的变量了。

```
MouseButtonDownProc mouse_button_handler;
```

也可以定义把 delegate 类型作为参数或者函数类型的返回值。

在表达式中，通过单独的函数名就可以创建函数类型的值。函数类型的值可以赋值给与其具有互换性的 delegate 类型的变量，还可以像函数的实际参数那样，作为参数进行传递。

```
void mouse_button_func(Window window, int x, int y, MouseButton button){
    ⋮
}

// 在 Window 对象中按下鼠标的时候调用
```

```
// 为了处理这个事件将 mouse_button_func 作为参数传递进去
window.set_mouse_button_down_proc(mouse_button_func);
```

delegate 类型的变量不仅可以赋函数，还可以赋方法。在赋方法的时候，delegate 类型的变量会将方法对应的类实例一并保存起来。

给 delegate 类型的变量赋值，需要满足以下条件。

1. 赋值的函数或者方法，必须与被赋值的 delegate 类型的参数数量一致。
2. 赋值的函数或者方法的参数类型（相对应地），必须与被赋值的 delegate 的参数类型一致，或者是 delegate 的参数类型的父类。
3. 赋值的函数或者方法的返回值类型，必须与被赋值的 delegate 的返回值类型一致，或者是 delegate 返回值类型的子类。
4. 赋值的方法在 throws 中列出的异常范围要比被赋值的 delegate 在 throws 中列出的范围小。

B.3.6 内建方法

在数组和字符串类型中，拥有一些内建方法。

数组的内建方法如下所示。

形式	含义
void add(T value);	向数组的末尾增加元素。T 的类型必须是可以赋值给数组元素的类型
int size();	获取数组的元素个数
void resize(int new_size);	改变数组的元素个数。如果新指定的大小小于当前数组的元素个数，则根据新的大小舍弃数组后面的元素。如果比当前大小大，则增加元素并把这些元素设置为该类型的默认值
void insert(int pos, T value);	在指定下标（pos）的元素前面插入 value 指定的值。T 的类型必须是可以赋值给数组元素的类型
void remove(int pos);	删除指定下标（pos）的元素，并将后面的元素向前移动一个下标。结果是数组的元素个数减少了 1 个

字符串的内建方法如下所示。

形式	含义
int length();	获取字符串的长度
`string substr(int pos, int len);`	截取字符串，并返回包含截取内容的新字符串。第 1 个参数指定开始截取的位置（第 1 个字符是 0），第 2 个参数指定要截取的长度

B.4 表达式

表达式（expression）是由运算符连接起来的单目表达式。

B.4.1 单目表达式

单目表达式有以下几种。

- 字面量。
- 标识符。虽然变量名、常量名、函数名是单目表达式，但类名等类型的名称不是表达式。
- `this`。表示在类的方法内指向当前实例的引用。
- `super`。在类的方法内，调用超类的方法时使用。
- `new` 表达式。具体的使用方法请参考 B.7.6 节。
- 枚举。

B.4.2 类型转换

在 Diksam 中，具备了以下条件时会进行自动转型。

■ 双目运算符的类型转换

双目算数运算符（`+`、`-`、`*`、`/`、`%`）以及比较运算符（`==`、`!=`、`>`、`>=`、`<`、`<=`）在两边类型不一致的时候，会根据以下规则进行类型扩展。

1. 无论左边还是右边，只要其中一边是整数（`int`），另外一边是实数（`double`）的时候，整数（`int`）会转换为实数（`double`）。

2. 在左边是字符串（`string`）的情况下，仅限于加法运算（运算符为 +）的时候，右边会转换为字符串。

上述类型转换，在算术赋值运算符（+=、-=、*=、/=、%=）的情况下同样适用。

■ 赋值时的类型转换

1. 把类赋值给类的时候，当左边是右边的超类或者被实现的接口时，会进行**向上转型**（up cast）。
2. 右边是 `int` 类型，左边是 `double` 类型的时候，右边会转换为 `double` 类型。
3. 右边是 `double` 类型，左边是 `int` 类型的时候，右边会转换为 `int` 类型。

B.4.3 运算符一览

Diksam 的运算符一览如下所示（优先顺序从高至低）。

<table>
<tr><th>运算符</th><th>含义</th></tr>
<tr><td>++ -- () [] . instanceof :: 类型 :</td><td>自增、自减、函数调用、引用数组元素、引用对象的成员、类的类型检查（与 Java 的设计相同）、向下转型</td></tr>
<tr><td>（单目）- !</td><td>符号反转、逻辑非</td></tr>
<tr><td>* / %</td><td>乘法、除法、模</td></tr>
<tr><td>+ -</td><td>加法、减法。加法运算符可以用于连接字符串</td></tr>
<tr><td>< <= > >=</td><td>比较大小。字符串也可以比较大小（设计以 C 语言的 strcmp() 为基准）</td></tr>
<tr><td>== !=</td><td>等值比较。字符串也可以进行比较（比较的不是引用而是值）</td></tr>
<tr><td>&&</td><td>逻辑与（AND）运算符。短路运算符在表达式 a && b 中，如果 a 为真，b 就不做判断了</td></tr>
<tr><td>||</td><td>逻辑或（OR）运算符。短路运算符，在表达式 a || b 中，如果 a 为真，b 就不做判断了</td></tr>
<tr><td>= += -= *= = %=</td><td>赋值运算符。含义与 C 语言中的相同</td></tr>
<tr><td>,</td><td>逗号运算符。按照从左到右的顺序计算表达式，返回右边表达式的值</td></tr>
</table>

在 Diksam 中，只存在后置的自增和自减运算符。并且，其动作与 C 等语言中前置的运算符效果相同（不会等到最后，而是立刻自增表达式的值）。

B.5 语句

B.5.1 声明语句

声明语句（declaration statement）是进行变量声明的语句。

声明语句为

```
类型 变量名；
```

或者

```
类型 变量名 = 初始化表达式；
```

这样的形式（“= 初始化表达式”的部分被称为**初始化**（initializer））。

为声明语句加上 `final`，可以防止对应变量的初始值被覆盖（禁止再次赋值）。

```
final int a = 10;
```

在 `final` 的声明语句中，如果不进行初始化就会发生编译错误。

使用 `const` 关键字声明常量。

```
const HOURS_IN_DAY = 24;// 一天 24 小时
const MINUTES_IN_DAY = HOURS_IN_DAY * 60;// 1 天 24 × 60 分
```

在常量的声明中，因为通过初始值可以判断类型，所以 `const` 无需指定类型。

B.5.2 表达式语句

所谓**表达式语句**（expression statement）就是在表达式后面加上分号。

```
// 调用 print() 函数的表达式后面加上分号
print("hello,world.");

// 自增表达式后面加上分号
i++;

// 无副作用的表达式也可以成为表达式语句，但没有任何意义
5;
```

B.5.3 if 语句

if 语句是根据条件判断并处理分支的语句。

```
if （条件表达式 1） {
    // 条件表达式 1 为真时执行的处理。
} elsif （条件表达式 2） {
    // 条件表达式 1 不为真，且条件表达式 2 为真时执行的处理。
} else {
    // 所有条件表达式都不为真时执行的处理。
}
```

elsif 子句和 else 子句都是可以省略的。另外，可以编写任意多个 elsif 子句。与 C 等语言不同的是，只包含一行语句也不能省略花括号（{}）。

B.5.4 switch 语句

switch 语句是进行多分支处理的语句。

```
switch(a)
case 1 {
    // a 等于 1 时执行
} case 2,3 {
    // a 等于 2 或 3 时执行
} default {
    // a 不等于上面的 1、2、3 时执行
}
```

Diksam 的 switch 语句与 C 语言的 fall through 不同。在匹配了一个 case 之后，即使没有在分支中写 break，也不会交由下一个 case 处理。另外，各 case 子句必须使用花括号（{}）括起来。

在针对多个值进行相同处理的时候，值之间以逗号分隔。

如果没有匹配任何一个 case 子句的话，就会执行 default 子句。

各 case 子句在以下代码成立时就会执行。

```
switch 中描述的表达式 == case 中描述的表达式
```

虽然只能使用 == 运算符，但是表达式的类型没有限制。然而，在 == 两边为

了匹配类型会进行类型转换，在 `switch` 中却不会进行类型转换。

B.5.5 while 语句

`while` 语句是进行循环操作的语句。

```
标识符 :
while ( 条件表达式 ) {
    // 在条件表达式为真的情况下，此处的代码会反复执行。
}
```

“标识符 :”的部分被称为**标签**(label)。标签可以省略。

与 `if` 语句相同，即使只包含一行语句也不能省略花括号（{ }）。

B.5.6 do while 语句

`do while` 语句是进行后判断型的循环操作语句。

```
标识符 :
do {
    // 这个位置的代码最初必须执行一次，
    // 之后，在条件表达式为真的情况下，此处的代码会反复执行。
} while ( 条件表达式 );
```

“标识符 :”的部分被称为**标签**(label)。标签可以省略。

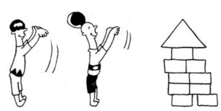
与 `if` 语句相同，即使只包含一行语句也不能省略花括号（{ }）。

B.5.7 for 语句

`for` 语句是进行循环操作的语句。

```
标识符 :
for ( 第 1 表达式 ; 第 2 表达式 ; 第 3 表达式 ) {
    // 在第 2 表达式为真的情况下，此处的代码会反复执行。
}
```

“标识符 :”的部分被称为**标签**(label)。标签可以省略。 上述 `for` 语句，除了在 `continue` 时会执行第 3 表达式之外，其他与下面的 `while` 语句效果相同。



```
第 1 表达式 ;
while ( 第 2 表达式 ) {
    # 语句
    第 3 表达式 ;
}
```

第 1 表达式、第 2 表达式、第 3 表达式都是可以省略的。在省略了第 2 表达式时，意味着永远返回真。

B.5.8 return 语句

return 语句是从函数中跳出的语句。

```
return 表达式 ;
```

在函数为 void 类型的时候，如果写了 return 表达式的话会报错。

B.5.9 break 语句

break 语句是用于跳出循环的语句。

```
break 标识符 ;
```

标识符可以省略。在省略的情况下，跳出最内侧的循环。

在指定了标识符的情况下，跳出持有同样标识符的循环。

B.5.10 continue 语句

continue 语句用于跳转到循环的末尾。

```
continue 标识符 ;
```

标识符可以省略。在省略的情况下，以最内侧的循环为对象。

在指定了标识符的情况下，以持有同样标识符的循环为对象。

在以 for 为对象进行 continue 的时候，continue 后会紧接着对 for 语句的第 3 表达式进行计算。

在以 do while 为对象进行 continue 的时候，会对下一次的条件表达式

进行计算，如果结果为假则终止循环。

B.5.11 try 语句

try 语句是用于执行异常处理的语句。

```
try {
    // 语句
} catch（异常类型 变量名）{
    // 语句
} catch（异常类型 变量名）{
    // 语句
} finally {
    // 语句
}
```

在 try 子句中如果发生了异常，就会执行 catch 子句，会在 catch 子句中按照从上到下的顺序搜索与发生的异常类匹配的 catch 子句。如果有匹配的 catch 子句，则会转移处理位置（抛给上一层或者终止处理）。发生的异常会被设置到 catch 子句声明的变量中。这个变量默认为 final，不可以再次赋值。

finally 子句无论异常是否发生，也不论是否有匹配的 catch 子句，都一定会执行。在 try 子句或者 catch 子句中即使是执行了 break、continue、return 来中断处理，也会执行 finally 子句。如果使用 break、continue、return 来中断 finally 子句的话，就会发生编译错误。

B.5.12 throw 语句

throw 语句是用于抛出异常的语句。

在通常情况下，可以像下面这样显式地为表达式指定异常。

```
throw e;
```

在 try 子句内发生的异常，如果存在与其对应的 catch 子句的话，就会被 catch 子句捕获。如果不存在与其对应的 catch 子句，或者异常发生在 try 语句之外的时候，会终止当前正在执行的函数并返回到调用者（或者叫

上一层）的处理中。如果调用者也没有 catch 这个异常的话，则会沿着调用层级回溯直到顶层结构。如果顶层结果中也没有捕捉这个异常的话，就会输出**栈轨迹**（stack trace）并终止处理。

Diksam 的栈轨迹在异常 throw 的时候会被清空，并在返回到调用函数的层级或者中断顶层结构执行的时候被再次设置。

因此，在 catch 子句中直接再次抛出捕获的异常时，如果不想清除当前的栈轨迹，可以像下面这样单独使用 throw;。

```
throw;
```



B.6 函数

B.6.1 函数定义

函数通过如下的形式进行定义。

```
类型 函数名（参数序列） throws 子句 {
    语句
}
```

throws 子句可以省略。throws 描述了在这个函数中有可能会发生的异常，用逗号分隔并一一列出。

如下例所示。

```
// 接收两个整数型参数并返回它们的和的函数
int sum (int a, int b) {
    return a + b;
}

// 以数组和索引值为参数，
// 返回指定位置的元素的函数。
int get_at(int[] array, int index)
  throws ArrayIndexOfBoundsException {
    return array[index];
}
```

函数的形式参数与已经赋值的 final 局部变量的使用方法一致。因此，不

能赋值给形式参数。

B.6.2 函数的签名声明

当函数被定义在其他源文件中的时候，就要像下面这样，以描述**签名声明**的方式，让调用了这个函数的表达式可以顺利编译。

```
int func(bouble x);
```

与 C 语言的原型声明不同，在 Diksam 中不能省略形式参数的名称（上面的 x）。另外，形式参数的名称也属于这个类型的一部分，在与函数定义不一致的时候就会发生错误。

例如下面的签名声明。

```
void draw_line(double x1, double y1, double x2, double y2);
```

与此相对，就不允许像下面这样的函数定义。

```
void draw_line(double x1, double x2, double y1, double y2);
```

B.6.3 局部变量

在函数内声明的变量就是**局部变量**(local variable)。

局部变量的作用范围是满足下面两个条件的范围（现在，局部变量的作用范围不受程序块的影响）。

- 在声明之后
- 在包含其声明的程序块内

局部变量的生命周期随着它所在函数的退出而终止。

B.7 类 / 接口的定义

B.7.1 类定义

类通过以下的形式进行定义。

```
类修饰符 class 类名
    : 超类，要实现的接口（多个）{
    字段、方法、构造方法的定义
}
```

类修饰符有以下两种。

- **访问修饰符**（access modifier）。可以指定为 public 或者不指定。
- **abstract 修饰符**。可以指定或者不指定。

访问修饰符与 abstract 修饰符的顺序不同。

当访问修饰符指定了 public 的时候，这个类就可以被别的包使用了。如果不指定的话，就只能在当前包中使用。

指定了 abstract 的类被视为**抽象类**（abstract class）。抽象类不能通过 new 将其实例化。相反，没有指定 abstract 的类被称为**具象类**（concrete class），其中不能包含 abstract 方法。

超类和要实现的接口与 C++、C# 相同，在冒号后面以逗号分隔并一一列出（顺序不同）。在不需要继承的情况下，可以省略冒号。

在 Diksam 中对于类来说只能单继承。另外，在 Diksam 中不能继承具象类。

接口可以被多继承。

B.7.2 接口定义

接口通过以下的形式定义。

```
访问修饰符 interface 接口名 {
    // 方法定义
}
```

接口除了只能记录 abstract 方法外，其他的功能与类相同。

访问修饰符与类相同，可以指定或者不指定 public。接口一定要加上 abstract，如果不指定 abstract 就会发生编译错误。

在 Diksam 中，接口不能继承其他接口。

B.7.3 字段定义

字段通过以下的形式定义。

```
访问修饰符 final 修饰符 类型 字段名 初始化语句（表达式）;
```

针对字段的访问修饰符有 public、private 和不指定。它们拥有不同的含义，public 可以在其他包中使用，不指定的话可以在本包内被使用，private 只能在当前类内被使用。

如果指定 final 修饰符，这个字段就不能被赋值了。

B.7.4 方法定义

方法通过以下的方式定义。

```
方法修饰符 类型 方法名（参数，可以是多个）throws 子句 {
    语句
}
```

与函数定义一样，throws 子句可以省略。另外，在加上了 abstract 修饰符后，可以不描述方法体（包含了语句的代码块）。

方法修饰符有以下几种。

- **访问修饰符**。public、private、不指定。
- **abstract 修饰符**。可以指定或者不指定。
- **virtual 修饰符、override 修饰符**。

没有指定 virtual 修饰符的方法不能重写。另外，如果要重写方法的话，必须指定 override 修饰符。

B.7.5 方法重写

在子类中定义与父类或者所继承接口的方法同名的方法被称为**方法重写**（method override）。方法重写必须满足以下条件。

1. 重写与被重写的方法，参数个数必须一致。
2. 进行重写的方法的参数类型（相对应地），必须与被重写方法的参数类型一致，或者是被重写方法的参数类型的父类。
3. 进行重写的方法的返回值类型，必须与被重写方法的返回值类型一致，或者是被重写方法的返回值类型的子类。
4. 进行重写的方法在 throws 中列出的异常的范围要比被重写方法在 throws 中列出的范围小。
5. 进行重写的方法的访问修饰符，要比被重写方法的访问修饰符的限制宽松许多。

B.7.6 构造器

构造器是在实例创建时自动调用的方法。

在 Diksam 中，构造器的定义需要使用 constructor 修饰符。

```
public constructor initialize(){
    // 记录构造器的处理
}
```

上面的例子用 constructor 修饰符描述了与类型名名称相同的构造方法，因此必须要在方法前加上 public 和 virtual 方法修饰符，以便在之后进行重写。

与 Java、C++、C# 不同，在 Diksam 中可以给构造器任意起名字。在 new 实例的时候，使用以下的形式指定构造器。

```
// myinit() 是用户自定义的构造器
Point p = new Point.myinit(x, y);
```

如果不指定方法名，只写成 new Point(x, y); 的话，会调用名为 initialize 的构造器。另外，在类定义的时候，如果一个构造器也没有定义的话，编译器会自动创建如下的**默认构造器**（default constructor）。

```
public virtual constructor initialize(){
    super.initialize();  ←仅限于存在超类的情况
}
```

Diksam 的构造器可以在子类中进行重写。但是，构造器并不会进行 B.7.5 节中写到的参数和返回值的检查。

另外，构造器仅限于在创建实例的时候使用，在实例被创建以后，构造器并不能像普通方法那样被调用。

B.8 程序库

在这里收录了 Diksam 标准程序库（library）中具有代表性的函数和类。

B.8.1 函数

`void print(string str);`

向标准输出中输出作为参数接收的字符串。

`void println(string str);`

向标准输出中输出作为参数接收的字符串，并在结尾处加上换行符。

`File fopen(string file_name, string mode);`

打开文件。参数的设计以 C 语言的 `fopen()` 为基准。

`string fgets(File file);`

从 `file` 指定的文件中读取一行，并作为返回值返回。

`void fputs(string str, File file);`

向 `file` 指定的文件中输出 `str`。

`void fclose(File file);`

关闭参数 `file` 指定的文件。

`double to_double(int int_value);`

将 `int` 转换为 `double`。

```
int to_int(double double_value);
```

将 double 转换为 int。

B.8.2 内建类

■ File

File 类相当于 C 语言的 File* 类型。其内部以 native_pointer 类型保存了 C 语言的 File*，因此并没有存在特别的方法等内容。

■ Exception

Exception 类是所有异常类的类层级的顶端。

它具有以下的字段和方法。

```
public string message;
```

保存了异常信息的字段。

```
public StackTrace[] stack_trace;
```

保存了栈轨迹的字段。

```
void print_stack_trace();
```

输出栈轨迹的方法。

另外，StackTrace 类的定义如下所示。

```
class StackTrace {
    int line_number;
    string file_name;
    string function_name;
}
```

附录 C Diksam Virtual Machine 指令集

本章将要展示 Diksam VM（DVM）的指令集一览表 *。

* 在语言的设计中似乎列出了一些不支持的指令，请把它们看作是不能保证执行正确性的隐藏功能。

C.1 范例

指令

DVM 指令的助记符。

操作数的类型

`byte` 为一个字节的正整数，`short` 为两个字节的正整数（大尾序），`cp` 指的是常量池的索引值，实际和 `short` 相同。

含义

表示当前指令的含义。

栈

表示指令执行时栈的变化。`[]` 内表示参与操作的栈顶值的类型。右端是操作后的栈顶。在 DVM 中没有 `boolean` 和 `function` 类型，实际上它们都是 `int` 值，只不过为了容易理解而写成了 `boolean`、`function`。

在没有给出运算符的一侧（箭头的左侧），顺序是有意义的，因此 `[int1 int2]` 的运算结果会被描述为 `[(int1 / int2)]`。结果的类型以 C 语言的运算符为基准，例如 `[(int1 > int2)]` 的类型为 `boolean`。

since

表示指令对应的是哪个版本。

C.2 指令一览表

指令	操作数类型	含义	栈	since
push_int_1byte	byte	将操作数指定的一个字节的整数入栈	[] → [int]	0.1
push_int_2byte	short	将操作数指定的两个字节的整数入栈	[] → [int]	0.1
push_int	cp	将常量池中的 int 常量入栈	[] → [int]	0.1
push_double_0		将 double 常量 0 入栈	[] → [double]	0.1
push_double_1		将 double 常量 1 入栈	[] → [double]	0.1
push_double	cp	将常量池中的 double 常量入栈	[] → [double]	0.1
push_string	cp	将常量池中的 string 常量入栈	[] → [string]	0.1
push_null		将 null 入栈	[] → [object]	0.2
push_stack_int	short	将栈中以 base 为基准以操作数为偏移量的位置的 int 值入栈	[] → [int]	0.1
push_stack_double	short	将栈中以 base 为基准以操作数为偏移量的位置的 double 值入栈	[] → [double]	0.1
push_stack_string	short	将栈中以 base 为基准以操作数为偏移量的位置的 string 值入栈	[] → [string]	仅 0.1
push_stack_object	short	将栈中以 base 为基准以操作数为偏移量的位置的 object 值入栈	[] → [object]	0.2
pop_stack_int	short	将栈中以 base 为基准以操作数为偏移量的位置的 int 值出栈	[int] → []	0.1
pop_stack_double	short	将栈中以 base 为基准以操作数为偏移量的位置的 double 值出栈	[double] → []	0.1
pop_stack_string	short	将栈中以 base 为基准以操作数为偏移量的位置的 string 值出栈	[string] → []	仅 0.1
pop_stack_object	short	将栈中以 base 为基准以操作数为偏移量的位置的 object 值出栈	[object] → []	0.2

（续）

指令	操作数类型	含义	栈	since
push_static_int	short	以操作数为索引值，将对应的 int 型静态变量入栈	[] → [int]	0.1
push_static_double	short	以操作数为索引值，将对应的 double 型静态变量入栈	[] → [double]	0.1
push_static_string	short	以操作数为索引值，将对应的 string 型静态变量入栈	[] → [string]	仅 0.1
push_static_object	short	以操作数为索引值，将对应的 object 型静态变量入栈	[] → [object]	0.2
pop_static_int	short	将栈顶的值出栈保存为 int 型静态变量，其索引值由操作数指定	[int] → []	0.1
pop_static_double	short	将栈顶的值出栈保存为 double 型静态变量，其索引值由操作数指定	[double] → []	0.1
pop_static_string	short	将栈顶的值出栈保存为 string 型静态变量，其索引值由操作数指定	[string] → []	仅 0.1
pop_static_object	short	将栈顶的值出栈保存为 object 型静态变量，其索引值由操作数指定	[object] → []	0.2
push_constant_int	short	将操作数指定索引值的 int 型常量入栈	[] → [int]	0.4
push_constant_double	short	将操作数指定索引值的 double 型常量入栈	[] → [double]	0.4
push_constant_object	short	将操作数指定索引值的 object 型常量入栈	[] → [object]	0.4
pop_constant_int	short	将栈顶的值出栈保存为 int 型常量，其索引值由操作数指定	[int] → []	0.4
pop_constant_double	short	将栈顶的值出栈保存为 double 型常量，其索引值由操作数指定	[double] → []	0.4
pop_constant_object	short	将栈顶的值出栈保存为 object 型常量，其索引值由操作数指定	[object] → []	0.4
push_array_int		根据栈中的数组和下标获取数组中的元素（int 型），并将其入栈	[array int] → [int]	0.2

（续）

指令	操作数类型	含义	栈	since
push_array_double		根据栈中的数组和下标获取数组中的元素（double 型），并将其入栈	[array int] → [double]	0.2
push_array_object		根据栈中的数组和下标获取数组中的元素（object 型），并将其入栈	[array int] → [object]	0.2
pop_array_int		将栈上的值（int1）赋值给数组 array 中下标为 int2 的元素	[int1 array int2] → []	0.2
pop_array_double		将栈上的值（double）赋值给数组 array 中下标为 int 的元素	[double array int] → []	0.2
pop_array_object		将栈上的值（object）赋值给数组 array 中下标为 int 的元素	[object array int] → []	0.2
push_character_in_string		从栈顶获取索引值，并取得栈中排在第二个位置的字符串，将字符串中与索引值对应（从 0 开始）的字符的字符编码入栈	[string int] → [int]	0.4
push_field_int	short	从栈中对象的字段中（由操作数指定索引值）取得 int 型的值并将其入栈	[object] → [int]	0.3
push_field_double	short	从栈中对象的字段中（由操作数指定索引值）取得 double 型的值并将其入栈	[object] → [double]	0.3
push_field_object	short	从栈中对象的字段中（由操作数指定索引值）取得 object 型的值并将其入栈	[object] → [object]	0.3
pop_field_int	short	将栈中 int 型的值出栈，并赋值给栈中指定对象的字段（由操作数指定索引值）	[int object] → []	0.3
pop_field_double	short	将栈中 double 型的值出栈，并赋值给栈中指定对象的字段（由操作数指定索引值）	[double object] → []	0.3

（续）

指令	操作数类型	含义	栈	since
pop_field_object	short	将栈中 object 型的值（object1）出栈，并赋值给栈中指定对象（object2）的字段（由操作数指定索引值）	[object1 object2] → []	0.3
add_int		进行 int 间的加法运算，并将结果入栈	[int int] → [int]	0.1
add_double		进行 double 间的加法运算，并将结果入栈	[double double] → [double]	0.1
add_string		进行 string 间的加法运算，并将结果入栈	[string1 string2] → [(string1 + string2)]	0.1
sub_int		进行 int 间的减法运算，并将结果入栈	[int1 int2] → [(int1 - int2)]	0.1
sub_double		进行 double 间的减法运算，并将结果入栈	[double1 double2] → [(double1 - dou-ble2)]	0.1
mul_int		进行 int 间的乘法运算，并将结果入栈	[int int] → [int]	0.1
mul_double		进行 double 间的乘法运算，并将结果入栈	[double double] → [double]	0.1
div_int		进行 int 间的除法运算，并将结果入栈	[int1 int2] → [(int1 / int2)]	0.1
div_double		进行 double 间的除法运算，并将结果入栈	[double1 double2] → [(double1 / double2)]	0.1
mod_int		进行 int 间的模运算，并将结果入栈	[int1 int2] → [(int1 % int2)]	0.1
mod_double		进行 double 间的模运算，并将结果入栈	[double1 double2] → [(fmod(double1, double2))]	0.1

（续）

指令	操作数类型	含义	栈	since
bit_and		按位进行与运算（int 之间），并将结果入栈	[int int] → [int]	0.4
bit_or		按位进行或运算（int 之间），并将结果入栈	[int int] → [int]	0.4
bit_xor		按位进行异或运算（int 之间），并将结果入栈	[int int] → [int]	0.4
minus_int		反转栈顶 int 值的符号	[int] → [int]	0.1
minus_double		反转栈顶 double 值的符号	[double] → [double]	0.1
bit_not		按位进行非运算，并将结果入栈。（将栈顶的 int 值按位取反）	[int] → [int]	0.4
increment		自增栈顶的 int 值	[int] → [int]	0.1
decrement		自减栈顶的 int 值	[int] → [int]	0.1
cast_int_to_double		将栈顶的 int 值转换为 double	[int] → [double]	0.1
cast_double_to_int		将栈顶的 double 值转换为 int	[double] → [int]	0.1
cast_boolean_to_string		将栈顶的 boolean 值转换为字符串（true 或者 false）	[boolean] → [string]	0.1
cast_int_to_string		将栈顶的 int 值转换为字符串	[int] → [string]	0.1
cast_double_to_string		将栈顶的 double 值转换为字符串	[double] → [string]	0.1
cast_enum_to_string		将栈顶的 enum 值转换为字符串	[enum] → [string]	0.4
up_cast	short	将栈顶指向对象的引用向上转型为操作数指定的类或者接口	[object] → [object]	0.3
down_cast	short	将栈顶指向对象的引用向下转型为操作数指定的类或者接口	[object] → [object]	0.3
eq_int		进行 int 间的比较（==）并将结果入栈	[int int] → [boolean]	0.1
eq_double		进行 double 间的比较（==）并将结果入栈	[double double] → [boolean]	0.1

（续）

指令	操作数类型	含义	栈	since
eq_object		进行 object 间的比较（==）并将结果入栈	[object object] → [boolean]	0.2
eq_string		进行 string 间的比较（==）并将结果入栈	[string string] → [boolean]	0.1
gt_int		进行 int 间的比较（>）并将结果入栈	[int1 int2] → [(int1 > int2)]	0.1
gt_double		进行 double 间的比较（>）并将结果入栈	[double1 double2] → [(double1 > double2)]	0.1
gt_string		进行字符串间的比较（>/ 字典顺序）并将结果入栈	[string1 string2] → [(wcscmp(string1, string2) > 0)]	0.1
ge_int		进行 int 间的比较（>=）并将结果入栈	[int1 int2] → [(int1 >= int2)]	0.1
ge_double		进行 double 间的比较（>=）并将结果入栈	[double1 double2] → [(double1 >= double2)]	0.1
ge_string		进行字符串间的比较（>=/ 字典顺序）并将结果入栈	[string1 string2] → [(wcscmp(string1, string2) >= 0)]	0.1
lt_int		进行 int 间的比较（<）并将结果入栈	[int1 int2] → [(int1 < int2)]	0.1
lt_double		进行 double 间的比较（<）并将结果入栈	[double1 double2] → [(double1 < double2)]	0.1
lt_string		进行字符串间的比较（</ 字典顺序）并将结果入栈	[string1 string2] → [(wcscmp(string1, string2) < 0)]	0.1

（续）

指令	操作数类型	含义	栈	since
le_int		进行 int 间的比较（<=）并将结果入栈	[int1 int2] → [(int1 <= int2)]	0.1
le_double		进行 double 间的比较（<=）并将结果入栈	[double1 double2] → [(double1 <= double2)]	0.1
le_string		进行字符串间的比较（<=/字典顺序）并将结果入栈	[string1 string2] → [(wcscmp(string1, string2) <= 0)]	0.1
ne_int		进行 int 间的比较（内容比较时使用 !=）并将结果入栈	[int int] → [boolean]	0.1
ne_double		进行 double 间的比较（内容比较时使用 !=）并将结果入栈	[double double] → [boolean]	0.1
ne_object		进行 object 间的比较（内容比较时使用 !=）并将结果入栈	[object object] → [boolean]	0.2
ne_string		进行 string 间的比较（内容比较时使用 !=）并将结果入栈	[string string] → [boolean]	0.1
logical_and		将逻辑与（AND）的结果入栈	[boolean boolean] → [boolean]	0.1
logical_or		将逻辑或（OR）的结果入栈	[boolean boolean] → [boolean]	0.1
logical_not		将栈顶的值取逻辑反（NOT）	[boolean] → [boolean]	0.1
pop		舍弃栈顶的一个值	[T] → []	0.1
duplicate		复制栈顶的一个值	[T] → [T T]	0.1
duplicate_offset	short	复制距离栈顶的第 n（操作数）个元素并将其入栈	[] → [object]	0.3
jump	short	跳转到操作数指定的地址	[] → []	0.1
jump_if_true	short	如果栈顶的值为 true，则跳转到操作数指定的地址	[boolean] → []	0.1

（续）

指令	操作数类型	含义	栈	since
jump_if_false	short	如果栈顶的值为 false，则跳转到操作数指定的地址	[boolean] → []	0.1
push_function	short	将操作数指定的函数的索引值入栈	[] → [function]	0.1
push_method	short	创建栈中对象中以操作数为索引值的方法入栈	[object] → [object int]	0.3
push_delegate	short	通过函数的索引值创建 delegate 并将其入栈	[] → [object]	0.4
push_method_delegate	short	创建栈中对象中以操作数为索引值的方法的 delegate，并将其入栈	[object] → [object]	0.4
invoke		调用栈顶的函数	[function] → [xx]	0.1
invoke_delegate		调用栈顶的 delegate	[object] → [xx]	0.4
return		以栈顶的值作为返回值并将函数 return	[返回值] → [xx]	0.1
new	short	对操作数 short 指定的索引值的类进行 new 操作	[] → [object]	0.3
new_array	byte,short	创建以操作数 short 所示类型组成的 byte 维数组（用栈中指定个数的元素），并将其入栈	[size1 size2 ...] → [array]	0.2
new_array_literal_int	short	以已经入栈的给定数量的 int 类型的操作数为元素创建数组，并将其入栈	[int1 int2 int3 ...] → [array]	0.2
new_array_literal_double	short	以已经入栈的给定数量的 double 类型的操作数为元素创建数组，并将其入栈	[double1 double2 double3 ...] → [array]	0.2
new_array_literal_object	short	以已经入栈的给定数量的 object 类型的操作数为元素创建数组，并将其入栈	[object1 object2 object3 ...] → [array]	0.2
super		将栈顶的对象引用的 vtable 转换为它父类的	[object] → [object]	0.3

（续）

指令	操作数类型	含义	栈	since
instanceof	short	返回栈顶的对象是否属于操作数的索引值所对应类的实例	[object] → [bool-ean]	0.3
throw		抛出栈顶的异常。同时清除栈轨迹	[object] → [xx]	0.4
rethrow		抛出栈顶的异常	[object] → [xx]	0.4
go_finally	short	把当前的程序计数器入栈，并跳转到操作数所示的地址	[] → [pc]	0.4
finally_end		在异常状态下throw捕获的异常。在非异常状态的时候，返回到通过go_finally跳转过来的位置（从栈中恢复程序计数器）	[pc] → []	0.4

编程语言实用化指南——写在最后

在本书中，我们一起制作了 crowbar 和 Diksam 两种编程语言。让我感到欣慰的是，书中的示例程序不只是停留在入门级别，而是达到了实用语言的水准。

亲爱的读者朋友们，希望你们也能尝试制作属于自己的编程语言。不过说出来你们可能会大跌眼镜，**编程语言的魅力基本上是由它的程序库来决定的**，而这是不容争辩的事实。

例如，Perl 由于正则表达式等强有力的字符串处理功能得到了广泛的应用。在 Perl4 的时候，作为编程语言，Perl 既没有引用，也不能创建数据结构，可以说很难用。但是，因为它处理文本文件十分方便，所以得到了广泛使用。同样，PHP 也因为提供了很多面向网页应用的功能而得到了普及。语言是否实用，是否能够普及，实际上和语言的设计本身没有太大关系。

因此，我在发明了 crowbar 和 Diksam 两种语言后，为它们加载了各自的程序库。

首先，我为 crowbar 加载了鬼车，使它具有用正则表达式处理文本的能力。

在 Diksam 中我用 crowbar 来处理文本。在文本处理这个领域里已经有了 Perl、Ruby 等语言，因此就算是为此特地制作一门语言也不会得到广泛普及。用来开发 Web 应用的编程语言更是琳琅满目，比如 PHP、Perl、Ruby、Java、ASP、ASP.NET 等，在这个领域中还充斥着各种框架，可以说是一个大杂烩。租赁服务器更是让人头疼，好不容易做的网页应用，有可能会因为服务器不能支持而不能使用。就这点来说，已经很让人沮丧了。

因此，我考虑让 Diksam 定位为“让初学者可以制作简单游戏的语言”。

在很早之前，我自己就是这样走上了编程的道路。

那个时候（1980 年左右）的个人电脑，大多将 BASIC 作为标准配置。那个时候的编程语言没有 `IF` 语句中 `begin~end` 或者 {} 之类的程序块的概念，选择分支的时候必须使用 `GOTO` 行号的方式进行跳转。也没有循环结构的 `FOR` 语句。要在循环外面记录循环计数器后，再使用 `GOTO` 进行跳转。虽然可以使用 `GOSUB` 制作子程序，但是不能定义局部变量，所有变量都要当做全局变量来处理。此外，变量名字不看到第 2 个字符，是区分不出来的 *。当然，这是时代的选择，不过，当时的 BASIC 作为编程语言来说还真是不怎么样。

* Visual Basic 的名字虽然继承了 Basic，但其实是完全不同的另一种语言。

即使如此，我当时只用了几十行代码就可以写一款射击游戏（用字符“⊥”

当做炮台来击落用字符“-o-”做成的飞碟）。我觉得这个过程十分有趣，这也是我踏上编程学习道路的第一步。

但是对于现在的年轻人来说，却不知道怎么去实现一款简单的游戏。

现在这些 PC 的性能与当时相比可以说有了飞跃性的提高，也出现了各种各样的编程语言和免费的处理器。但是，比如在 C 语言中，使用不依赖处理器的标准 C，就连窗体都打不开。即便只是在 Windows 中能够运行起来的程序，C 语言也要通过 Windows 的 API 来创建窗口，如此复杂的程序初学者根本应付不来。

我觉得在现在 C 语言的入门书中，多半从“`hello,world.`”开始介绍许多命令行程序。但是，我在中学时代，从最开始就能写出和“`hello,world.`”差不多的程序，接着就编了猜数字游戏，然后就想着要做一款打飞碟的游戏了。当今，计算机已经十分先进，但人们在这个方面反而退化了。

当然，现在不仅可以使用 C 语言，也考虑使用 Java。Java 中的 GUI 可以不依赖于处理器，因此也可以把游戏做成 Applet 发布在网站上，在朋友面前炫耀一把。但这样一来，在创建类的时候就需要继承一种叫作 `java.applet.Applet` 的类，并重写它的 `init()` 和 `paint()` 之类的方法。这里突然出现了很多面向对象的知识，初学者一时之间很难接受。也许有人会觉得我又在这里老调重弹了，但是仔细想想，为了从 GUI 接收输入，就必须要创建事件监听器、实现特定的接口，除此之外还需要使用内部类和匿名类。这些还没完，因为制作的是实时游戏，为了实现动画效果还需要使用多线程进行异步处理。这些对于一个新手来说简直是个噩梦。

再者，制作“打飞碟”这样一款游戏对于 JavaScript 来说也不是很容易。可以制作 FLASH 的语言 ActionScript，它的标准处理器又不是免费的。

HSP（Hot Soup Processor）语言是我在中学时代玩过的类似于 BASIC 的语言。不过，很对不起这门语言的 fans，这门语言本身和与它同时代的 BASIC 如出一辙，因此对于刚开始学习编程的人来说非常不推荐。它甚至没有 GOSUB*。

* 从 HSP3 开始可以通过函数实现。

因此，我在 Diksam 中加载了可以让“打飞碟”游戏实现起来更为简单的程序库 *。

* 这种情况仅限于 Windows 平台。

因为要制作的游戏非常简单，所以无须特意想着面向对象和事件驱动的概念。例如“打飞碟”游戏可以写成下面这样。

代码清单 1-4
“打飞碟”游戏的程序
(ufo.dkm)

```
require diksam.window;

// 创建设置窗体属性的 WindowAttribute 对象。
WindowAttribute attr = create_window_attribute();
// 设置背景色为黑色。
attr.background = create_solid_brush(0, 0, 0);

// 创建窗体。如果使用默认设置就可以，
// 那么不创建 attr 传入 null 即可。
Window w = create_window( “UFO 游戏”, 800, 600, attr);

// 使用 x 键终止程序。
w.set_destroy_proc(window_destroy_and_exit);
// 显示窗体。
w.show();

// 为了描绘窗体取得 Graphics 接口。
Graphic g = w.get_graphics();
// 将字符的背景色设置为黑色。
g.set_background_color(new Color(0, 0, 0));

// 战车、激光、UFO 的颜色设置。
Color tank_color = new Color(60, 255, 100);
Color ufo_color = new Color(60, 255, 255);
Color beam_color = new Color(255, 255, 100);
// 生成字体。详细的设置 ( FontAttribute )
// 与 WindowAttribute 相同，当前默认为 null。
Font font = create_font(25, null);

// 随机数的初始化
randomize();

// 游戏结束后再开始使用的循环
for(;;){
    // 设定炮台 ( tank )、ufo 的坐标。将 tank_x，ufo_x，ufo_y 的
    // 当前坐标赋值给 prev，作为前一次绘制的坐标 ( 消除时使用 )。
    // ufo_next_x,y 作为 ufo 的移动目标的坐标。
    // ufo 将在 ufo_next_x,y 的附近移动，
    // 但如果两次坐标基本相同，则重新设定 ufo_next_x,y。
    int tank_x = 0;
    int ufo_x = 0;
    int ufo_y = 0;
    int ufo_prev_x = ufo_x;
    int ufo_prev_y = ufo_y;
    int ufo_next_x = random(700);
    int ufo_next_y = random(450);
    // 是否存在炮台发射的激光的标识和激光坐标
```

```
    boolean beam_flag = false;
    int beam_prev_y;
    int beam_x;
    int beam_y;

    // 游戏的主循环
    for(;;){
        // 消除前一次画出来的UFO。
        g.draw_string(font, ufo_color, ufo_prev_x, ufp_prev_y,"     ");
        // 绘制UFO。
        g.draw_string(font, ufo_color, ufo_x, ufp_y," ├○┤ ");
        // 为了再次消除，保存本次绘制的坐标。
        ufo_prev_x = ufo_x;
        ufo_prev_y = ufo_y;
        // 绘制炮台。
        g.draw_string(font, ufo_color, tank_x, 540," /┴\ ");
        // 发射了激光的话……
        if(beam_flag){
            // 消除前面的激光，重画新的激光。
            g.draw_string(font, ufo_color, ufo_prev_x, ufp_prev_y," ");
            g.draw_string(font, ufo_color, ufo_x, ufp_y,"|");
            beam_prev_y = beam_y;

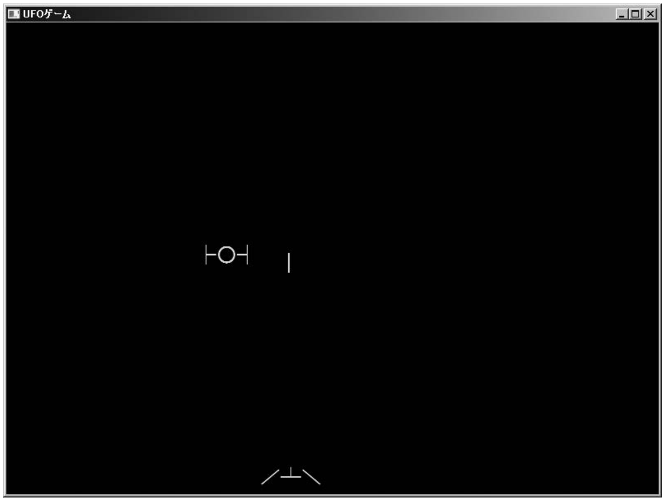
            // 碰撞判断。可能很幼稚。
            if(beam_x >= ufo_x && beam_x < ufo_x + 80
                && beam_y >= ufo_y && beam_y < ufo_y +60){
                // 被激光打中后跳出循环。
                break;
            }
        }

        // 通过判断键盘输入移动炮台。
        if(is_key_pressed(KeyCode.LEFT) && tank_x > 0){
            tank_x -= 10;
        } elseif(is_key_pressed(KeyCode.RIGHT) && tank_x < 700){
            tank_x += 10;
        }
        if(is_key_pressed(KeyCode.SPACE) && !beam_flag){
            beam_flag = true;
            beam_x = tank_x + 40;
            beam_y = beam_prev_y = 480;
        }
        // UFO的移动。在ufo_next_x,y的附近移动。
        if(ufo_x < ufo_next_x -10){
            ufo_x += 10;
        } elseif(ufo_x > ufo_next_x + 10){
            ufo_x -= 10;
```

```
        } else {
            // 如果两次坐标基本相同，则重新设定目标坐标。
            ufo_next_x = random(700);
        }
        if(ufo_y < ufo_next_y - 10) {
            ufo_y += 10;
        } elseif (ufo_y > ufo_next_y + 10) {
            ufo_y -= 10;
        } else {
            ufo_next_y = random(450);
        }
        // 激光的移动
        if(beam_flag) {
            if(beam_y < -20){
                beam_flag = false;
            }
            beam_y -= 20;
        }
        // 定时消息循环。无论有没有
        // 鼠标或者键盘事件，都等待 20 毫秒。
        timed_message_loop(w, 20);
    }

    // 被激光打中后跳出循环，执行这里。
    for(;;){
        // 显示爆炸效果
        Color explosion_color = new Color(255, 0, 0);
        g.draw_string(font, explosion_color, ufo_x, ufo_y, "***");
        timed_message_loop(w, 100);
        g.draw_string(font, explosion_color, ufo_x, ufo_y, "###");
        timed_message_loop(w, 100);
        // 按 N 重启游戏，按 Q 退出。
        if(is_key_pressed(KeyCode.N)){
            Brush b = create_solic_brush(0, 0, 0);
            g.fill_rectangle(b, 0, 0, 800, 600);
            b.dispose();
            break;
        } elsif (is_key_pressed(KeyCode.Q)) {
            exit(0);
        }
    }
}
```

游戏的截屏如下所示。



这是个跟我同时代的人都会怀念的画面吧 *。

Diksam 在这个领域的进化旅程才刚开始，谁也不知道它在未来会变成什么样子。但是，在代码清单“ufo.dkm”的程序中，只能有一架 UFO，在同一时间炮台只能发射一发激光（因为表示 UFO 和激光位置的变量只有一组）。如果觉得这样没意思的话，就必须要使用数组了。x 坐标和 y 坐标要是都使用数组来管理的话，肯定会很不方便，如果有类的话感觉就会方便很多。同样，即使不同时出现多个飞碟，如果想要各种各样的敌人轮番登场，就要使用继承和多态了……一门语言因为追寻着这样的思路而具有了面向对象的概念，真让人兴奋。

各位读者朋友，你们想让自己的编程语言向哪个方向发展呢？希望本书能给各位带来一些启发。

* 当然，在 Diksam 中也可以很简单地使用图标来表示 UFO。

参考文献

■ 书中引用到的文献

[1] 情報処理シリーズ７　コンパイラ

英文版：*Principles of Compiler Design*①

[2] プログラミング言語 C

英文版：*The C Programming Language*

中文版：《C 程序设计语言》（机械工业出版社）

[3] The Problem with Integer Division

http://python-history.blogspot.com/2009/03/problem-with-integer-division.html

[4] 文字符号の歴史―欧米と日本編

中文译名：《文字符号的历史——欧美与日本篇》

作者：安冈孝一 安冈素子

出版社：共立出版社，2006 年

[5] プログラミング作法

英文版：*The Practice of Programming*

中文版：《程序设计实践》（机械工业出版社）

[6] YARV アーキテクチャ

英文版：*YARV Architecture*

http://www.atdot.net/yarv/yarvarch.en.html

[7] The Java® Language Specification

http://docs.oracle.com/javase/specs/jls/se7/html/index.html

---

① *Principles of Compiler Design* 的封面是一名骑士和一只恐龙，因此被人称为“龙书”，但因为那条龙是绿色的，所以称为“绿龙书”。9 年后（1986 年），原来的两位作者加上 Ravi Sethi，升级了这本书，书名改为 *Compilers: Principles, Techniques and Tools*，封面依然沿用骑士和恐龙，那头龙是红色的，因此被叫作“龙书二”或者是“红龙书”。——译者注

[8] オブジェクト指向入門 第 2 版 原則· コンセプト

英文版：*Object-Oriented Software Construction,Second Edition*

[9] Effective Java プログラミング言語ガイド

英文版：*Effective Java*

中文版：《Effective Java 中文版（第 2 版）》（机械工业出版社）

[10] オブジェクト指向における再利用のためのデザインパターン

英文版：*Design Patterns: Elements of Reusable Object-Oriented Software*

中文版：《设计模式：可复用面向对象软件的基础》（机械工业出版社）

[11] The Trouble with Checked Exceptions

http://www.artima.com/intv/handcuffs.html

[12] Java の理論と実践：例外をめぐる議論 チェックすべきか、チャックせずにおくべきか

中文译名：Java 理论与实践：关于异常的争论 要检查，还是不要检查？

http://www.ibm.com/developerworks/cn/java/j-jtp05254/

[13] Exceptions

http://joelonsoftware.com/items/2003/10/13.html

[14] Cleaner, more elegant, and harder to recognize

http://blogs.msdn.com/b/oldnewthing/archive/2005/01/14/352949.aspx

[15] *More Joel On Software*

中文版：《软件随想录：程序员部落酋长 Joel 谈软件》（人民邮电出版社）

[16] Python リファレンスマニュアル 2.4.1. 文字列リテラル

英文版：*The Python Language Reference 2.4.1. String literals*

http://docs.python.org/2/reference/lexical_analysis.html#literals

■ **其他推荐的书**

关于制作编程语言的书虽然经常会出，但是多数由于出版量不大而慢慢地绝版了……（希望本书不会这样）

因此我将以前学习过的书作一个介绍（说句题外话，近藤嘉雪老师的“yacc によるCコンパイラプログラミング”（《使用yacc开发C语言编译器》）一书在日本亚马逊网站上卖到了245 000日元[①]，即使是这样我仍然觉得这本书很值得），在这里只介绍一些价格合适并且在市面上可以买到的书。

下面列出的是本书出版时（2009年4月）的参考书籍。

● 新コンピュータサイエンス講義 コンパイラ

中文译名：《新计算机科学讲座：编译器》

作者：田中育男

出版社：Ohmsha出版社，1995

日本编译器第一人田中育男先生的书。

田中先生的这本书以理论为主，难点很多。书中记录了类似于Pascal的语言处理“PL/0”的全部代码，很有实用价值。PL/0是一个递归下降语法分析器，因此本书可以为不想使用yacc来制作编程语言的人提供参考。

● lex & yacc プログラミング

英文版：*lex & yacc*

中文版：《lex与yacc》（机械工业出版社 已绝版）

O'Reilly的动物系列图书。我认为就凭它在这个系列中，这本书就值得信赖。它的出版时间较早，但可以作为yacc/lex的参考手册，是一本非常有实用价值的书。

下面这些书是我目前正在学习的。

● コンパイラ I －原理・技法・ツール

英文版：*Compilers: Principles, Techniques, and Tools (2nd Edition)*

中文版：《编译原理》（机械工业出版社）

可以说是编译器制作方面的圣经之著。因为封面上印有龙的图案所以被叫作“龙书”（确切地说应该是“红龙书”）（第4章中译者也引用了这本书中的内容）。

本书是原版的第一卷。现在，第二卷已经很难见到了（即便是作者这样的资深人士也没有机会收藏）。

本书内容不太容易理解（例如在不使用yacc的情况下制作LR解析器），但

① 相当于人民币15 000元左右。——译者注

我觉得这是本不可不读的好书。

● コンパイラの構成と最適化

中文译名:《编译器的结构与优化》

作者：田中育男

出版社：朝仓书店，1999

又是田中先生的书。其中对“优化”的 讲解占到了本书一半以上的篇幅。

如果你想要了解现在的商业化编译器是怎样做的，这本书再合适不过了。

● *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*

作者：Richar Jones,Rafael Lins

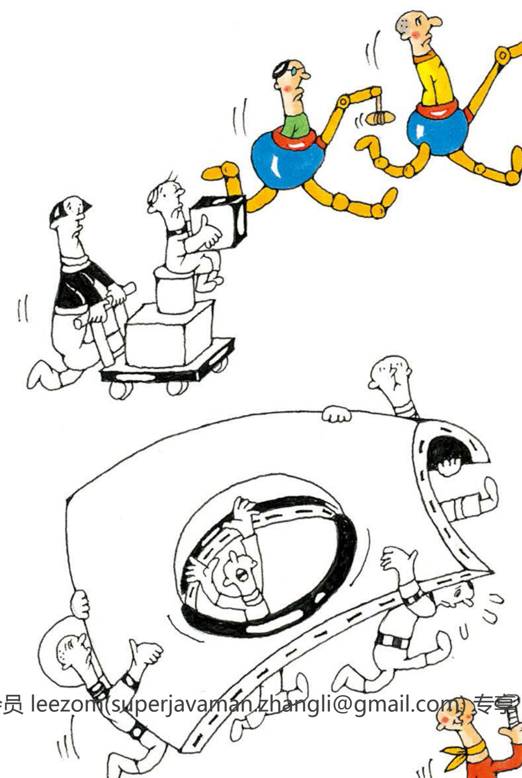
出版社：John Wiley & Sons，1996

尚无译本。本书中不仅介绍了标记 - 清除 GC 和 Copying GC 等算法，还讲解了通过简单地实现来解决问题的方法（分代式 GC、增量 GC、并发 GC 等）。

刘卓 2004年开始对日软件开发工作，其间还从事技术及软件工程相关培训工作。自2011年开始从事电力行业产品研发。持续关注企业级应用架构和Web客户端技术。

徐谦 6年技术开发及项目经验，曾以技术工程师身份赴日本工作两年，后归国联合创办互联网公司，现居上海继续创业中。主要从事PHP方向的Web开发。热爱开源，曾向Zend Framework等知名PHP开源项目贡献代码，并于Github自主研发运维EvaThumber等开源项目，获得国内社区认可。乐于分享技术心得，个人技术博客avnpc.com在国内PHP圈小有影响。

吴雅明 13年编程经验，其中7年专注于研发基于Java EE和.NET的开发框架以及基于UML 2.0模型的代码生成工具。目前正带领团队开发云计算PaaS平台及云计算自动化配置部署的系统。译著有《征服C指针》等。



图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
分类建议 计算机/编程语言
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-33320-9
定价：79.00元



看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks